AN EXECUTABLE FORMAL FRAMEWORK FOR REGULAR STRING TRANSFORMATION IN REWRITING LOGIC

BY

SHADI AYMAN ALHAJ

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

In

INFORMATION AND COMPUTER SCIENCE

MAY 2017

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **SHADI AYMAN ALHAJ** under the direction of his thesis adviser and approved by his thesis committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN INFORMATION AND COMPUTER SCIENCE**.

Thesis Committee

Dr. Musab A. Alturki (Adviser)

Dr. Mohammad Alshayeb (Member)

Dr. Wasfi Al-Khatib (Member)

Dr. Khalid Aljasser
Department Chairman

Dr. Salam A. Zummo
Dean of Graduate Studies

Date 8/6/2017

To my beloved Mom, Tharwat.

He who is afraid of climbing mountains will live all his life between the holes in the ground.

ACKNOWLEDGMENTS

I, first and foremost, submit my thankful praises to Allah.

"One can pay back the loan of gold, but one dies forever in debt to those who are kind and provide help" $\sim$ Malayan Proverb.

I would like to send my sincere gratitude to my mother for always endeavoring to keep me happy, safe, protected and nourished. Thank you for your kindness, caring and immeasurable patience. Also, Thank you for looking out for me even when I didn't think that I needed it and for letting me fall when I had to learn by making mistakes.

Dear Dr. Musab Alturki, I would like to express my sincere gratitude for all the efforts you have done during this academic journey and especially druing my master thesis; it has been amity.

I would also like to utilize this opportunity to acknowledge, appreciate and express my deep sense of gratitude towards all the faculty and staff members who have always replenished my mind and provided me with incredible insight about various courses and projects from my first day in KFUPM until graduation. It was a truly wonderful learning experience and an honor to learn from some of the very best teachers.

I am deeply thankful to all of my KFUPM friends and especially to my best

friend, neighbor, and my brother **Eng. Hamza Ahmed Mir** from Pakistan. I feel lucky and proud that Hamza is not only one of my best friend but has always supported me throughout this journey as he is smart, brilliant, clever, creative and a thorough gentleman. Also, I would like to thank **Eng. Anas Abdulhadi, Eng. Muath Alharithi,** and **Eng. Yaser Alharithi** for their support and help during this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

NAME: Shadi Ayman Alhaj

TITLE OF STUDY: An Executable Formal Framework for Regular String Transformation in Rewriting Logic

MAJOR FIELD: Information and Computer Science

DATE OF DEGREE: May 2017

Finite strings constitute a fundamental data type found in most general purpose programming languages. Furthermore, regular string transformations, which are a class of functions on finite strings, are widely used and have various applications in security, bioinformatics, etcetera. Given the ubiquity of regular string transformation and their importance, it would be useful to formally reason about these transformations at a high-level of abstraction, which is close to the application in which they are used.

In this thesis, we develop a formal framework based on rewriting logic and Maude in which regular string transformation can be formally specified and analyzed. We will also follow two complementary approaches to develop the framework that is executable and enable different types of formal analysis, such as simula-

tions, inductive theorem proving and reachability analysis for both deterministic and non-deterministic regular string transformations. One approach was theoretical, in which the formal semantics of the DReX language will be developed. DReX is a language for describing regular string transformations. The second approach is experimental, in which a corresponding executable specification in Maude will be developed and used for formal analysis. As a result, we develop: (i) an algebraic deterministic semantics of DReX in Maude, and (ii) an extended rewriting semantics of a non-deterministic generalization of DReX capturing non-deterministic regular string transformations. The approach is illustrated using several real-world examples and case studies.

ملخص الرسالة

الإسم: شادي أيمن الحاج.

عنوان الرسالة: إنشاء إطار تنفيذي يعتمد على العمليات الرياضية للتعامل مع طرق تحويل شكل النص.

التخصص: علوم الحاسب الآلي والمعلومات.

تاريخ الدرجة العلمية: أيار – 2017.

تشكل وحدة بيانات النصوص (Strings) أحد أنواع البيانات الأساسية في معظم لغات البرمجة. وعلاوة على ذلك، يوجد العديد من البرامج والتطبيقات الشائعة التي تستخدم عمليات معالجة وتحويل النصوص (Regular String Transformations)، حيث يتم إستخدامها في أمن المعلومات وفي المعلوماتية الحاسوبية (Bioinformatic) وغيرها. ونظراً لإنتشار إستخدام هذه العمليات بالإضافة الى أهميتها في التطبيقات المستخدمة سيكون من المفيد دراستها بأسلوب رسمي (Formally reason) على مستوى عال من التجريد دون الخوض في التفاصيل وبحيث تكون هذه الدراسة مبينة على التطبيقات التي تسخدم عمليات تحويل ومعالجة النص.

ومن خلال هذه الأطروحة قمنا بإستخدم لغة ال (Maude) كلغة برمجية - وهي لغة تعبير رياضية - لبناء وتطوير إطار رسمي (Formal Framework) على أساس إعادة كتابة المنطق (Rewriting Logic) لمعالجة وتحليل عمليات تحويل النصوص بالإضافة لإستخلاص وتحليل بعض الخصائص المتعلقة بعمليات معالجة وتحويل النصوص. ولتحقيق هذا الهدف قمنا بإتباع نهجين تكميليين لبناء إطار تنفيذي يسمح لنا القيام بأنواع مختلفة من التحليل الرسمي (Formal Analysis) مثل محاكاة عمليات معالجة وتحويل النصوص، تحليل نظرية الإستقراء الرياضي (Inductive Theorem Proving)، بالإضافة لتحليل إمكانية الوصول (Reachability Analysis) لكل من عمليات معالجة وتحويل النصوص المنتظمة وغير المنتظمة (Deterministic & Non-deterministic Regular String Transformations)

CHAPTER 1

INTRODUCTION

From airline reservations to the deployment of probes and satellites to celestial bodies, programming languages have become an important aspect of technology utilized in our daily lives. They also form a crucial part of computing and its related sciences. Finite strings are essential elements of program semantics which constitute a fundamental data type found in most general purpose programming languages [2]. String transformations describe the process of mapping strings from an input alphabet Σ to strings in an output alphabet Γ. String transformations are widely used and have various applications in security, natural language processing, data mining, bioinformatics, and so forth [3]. They are crucial tools for applications such as translating data from one format to another, spelling error correction, reformatting documents, sanitization of web addresses and other applications [3, 1, 4].

Regular string transformations (RSTs) are defined as a particular class of partial functions that transform a string into another string, which have a robust the-

oretical foundation including closure properties, multiple characterizations, and decidable analysis questions [1].

The following examples illustrate some uses of regular string transformations:

- Conversion of the lowercase to the uppercase character ($a \longrightarrow A$) could be represented using a partial function given by $\mathsf{f} : \mathsf{X} \rightharpoonup \mathsf{Y}$ mapped from a domain X to a codomain Y, where $[\mathsf{a..z}]$ and $[\mathsf{A..Z}] \subset \mathsf{UNICODE}$ represent the domain of X and Y respectively. For instance, in this mapping, according to UNICODE, 32 is subtracted from an ASCII code of lowercase letter "a" to convert it to uppercase.

- Web address sanitization is a regular string transformation process that eliminates or encodes dangerous characters in untrusted data [5]. For instance, `<h1>Hello World!</h1>` $\longrightarrow$ `Hello World!`. In this example, the web sanitizer removes tags or encodes special characters from this string.

- Code obfuscation is a transformation approach that has been used by developers to protect source codes, hide internal implementation details [6] and prevent software piracy. This string transformation generates a concealed string using a code that is abstruse for humans. For example, `int x = 5;` $\longrightarrow$ `/*\u002f\u0069\u006e\u0074 \u0020\u0078\u0020\u003d\u0020\u0035\u003b\u002f\u002a*/`.

Formally, regular string transformations are typically captured using either an appropriate logical system such as monadic second order (MSO) logic or some form of a finite state machine such as string transducers, streaming string transducers,

or two-way transducers, among others [1, 7]. Moreover, an alternative way to capture regular string transformations is by using domain-specific programming languages such AWK and Perl [8].

Definition 1.1 ([4, 9]) *The transducer generates an output string as the concatenation of all the obtained output symbols which are prior produced at each step corresponding to a given input string which can be presented as* $(\mathsf{Q}, \Sigma, \Gamma, \mathsf{I}, \mathsf{F}, \delta)$ *where:*

- Q *the set of states.*
- Σ *is the input alphabet;*
- Γ *is the output alphabet;*
- $I \in Q$ *the initial state;*
- $F \subseteq Q$ *is the set of final states;*
- δ *the state transition function, which maps* $Q \times \Sigma$ *to* Q*.*

1.1 Problem Description

Logical systems and finite state machines which are traditionally being used to capture regular string transformations, as described earlier, do not necessarily highlight the essence of regular string transformations. Their foundational features are inter-mixed with their description in the underlying formalisms. Consequently, formal analysis of such a transformation is usually done manually or

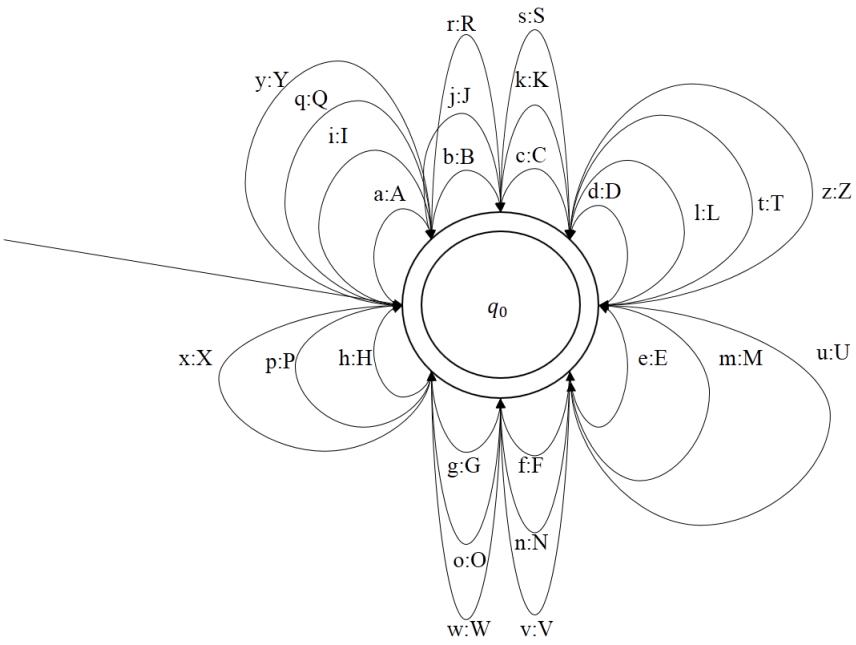

Figure 1.1: Convert Lower to Upper letters Finite Transducer

through complex unnatural encodings and usually results coupling of transducer semantics [8]. For example, the finite transducer machine that converts lowercase to uppercase characters as illustrated in Figure 1.1 shows the complexity of the transducer even for a simple regular string transformation. Moreover, string manipulation in programming languages such as Perl and AWK have been used to query and reformat text files for many years. We can observe that these languages have different compilers to capture regular string transformations.

Given the ubiquity of regular string transformations and their importance, it would be useful to think about these transformations at a high level of abstraction, which is close to the application in which they are used. In general, programming languages that have been heavily used for string transformations do not typically have a formal semantics.This shortcoming of the programming

languages motivated us to devise a formal and executable semantics to capture regular string transformation using the Maude framework. This formal definition can be used in various domains that utilize regular string transformations, such as bioinformatics, web security, and other domains.

This thesis, therefore, focuses on the formal specification and analysis of *deterministic* and *nondeterministic* regular string transformations. Having the motivation described above in mind, we formulate the research questions to be addressed by this thesis.

Research Questions:

- [**RQ1**] How do we formally specify *deterministic* regular string transformations in a way that is high-level, elegant and that can be used directly to evaluate and analyze their properties?

- [**RQ2**] How do we define an extension of this formal specification to provide a formal model for *nondeterministic* regular string transformations in an elegant high-level manner that enables us to simulate and analyze their properties?

The thesis aims to show that these problems can be appropriately addressed using an **algebraic semantics approach** to regular string transformations that will provide an elegant formal method for the specification of such transformations highlighting its essential features, and enabling directly their evaluation and formal analysis. Using an appropriate formalism based on **rewriting logic**, this method can also be extended to formal semantics of non-deterministic regular

string transformations while still enabling formal analysis of their properties.

1.2 Overview of the Approach

The methodology of the thesis is described graphically in Figure 1.2. We follow two complementary approaches. The first one is the theoretical development where we take the description of the language (DReX), a language for describing regular string transformations, and develop (1) a formal algebraic semantics in the form of an equational theory in order-sorted equational logic, and (2) an extension of this theory into rewrite theory in rewriting logic that captures non-deterministic semantics of more general regular string transformation.

The experimental development is the second approach that we follow in the thesis. Here, we develop specifications corresponding to the formal semantics above as Maude modules, since modules in Maude are executable and the specification can be directly used to perform different types of experiments. This includes simulations and formal analysis using several real-world examples and case studies.

1.3 Summary of Contributions

This work contributes a formal framework using rewriting logic and Maude that captures both *deterministic* and *nondeterministic* regular string transformations. The following list highlights the main research contributions of the thesis:

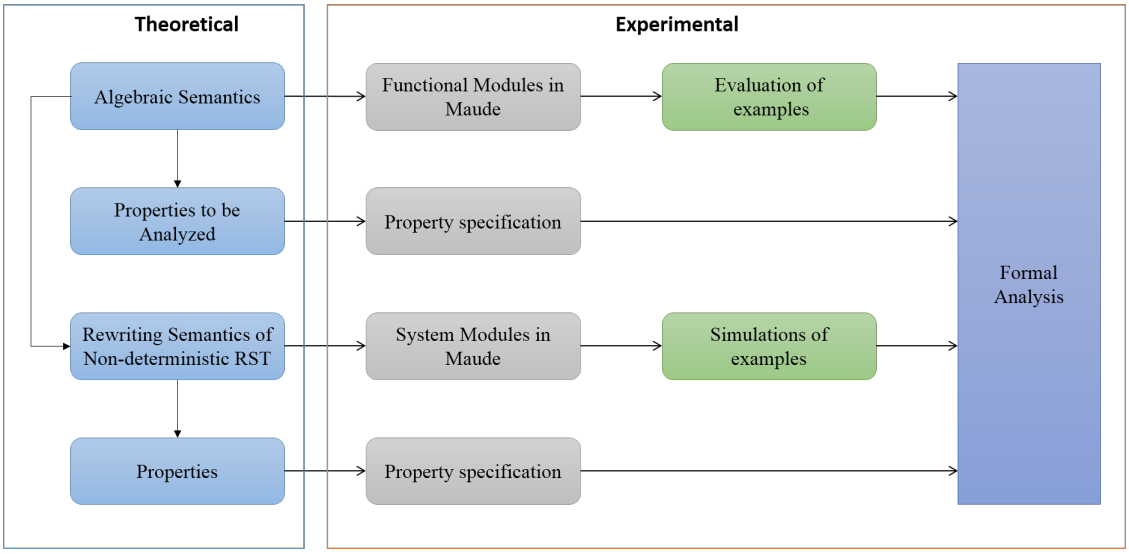

Figure 1.2: Thesis Methodology

1. A high-level formal algebraic semantics of regular string transformation.

2. An executable specification in Maude enabling formal analysis of deterministic regular string transformation.

3. A novel rewriting semantics that elegantly captures the non-deterministic behavior of regular string transformation process.

4. An effective mechanism for simulating and formally analyzing various of non-deterministic regular string transformations.

5. Formulation of a function library consisting of various formally verifiable regular string transformation functions that can be employed in numerous domains such as security encryption and web address sanitization.

1.4 Thesis Outline

This section gives a brief outline of the thesis.

The thesis begins in Chapter 2 by reviewing preliminaries on rewrite theory, Maude system, and DReX. The chapter also describes at a high level how such units of specifications in rewriting logic can be executed and analyzed in Maude. In Chapter 3, an algebraic formal specification has been proposed to capture deterministic regular string transformation. In Chapter 4, we discuss the results of evaluating regular string transformation programs and show examples of formal analysis. Chapter 5 presents the algebraic specification that has been extended to capture non-deterministic regular string transformation based on rewriting logic. Finally, a discussion of related work is given in Chapter 6, followed by a discussion of further research directions in Chapter 7.

CHAPTER 2

BACKGROUND

In this chapter, we review at a high level, some preliminaries on rewriting logic, the rewriting logic engine Maude and the DReX language. More details are available in the cited references below.

2.1 Rewriting Logic

Rewriting logic is a type of computational logic that generically includes both logical deduction and concurrent computation [10]. It has been used to provide a formal explanation of systems, programming languages, and has been used as a universal framework for language definition using various semantics styles such as structural operational semantics (SOS), continuation based semantics and reduction semantics [11]. Consequently, rewriting logic has proved to be a natural formalism to define the executable semantics of concurrent system and programming languages [12, 13]. In fact, rewriting logic specifications provide in practice, a simple way to develop executable formal definitions of languages [14].

A specification in rewriting logic is a rewrite theory. A rewrite theory $\mathcal{R} = (\Sigma, E, R)$ consists of (i) the equational theory (Σ, E) that specifies the semantics of a languages ***deterministic*** computations [15], the static aspect of a system and the distributed states of such system, where Σ is a collection of declarations of sorts, subsorts, and function symbols, and E specifies the algebraic identities, and (ii) a set of rewrite rules R that specifies the ***nondeterministic*** computations (concurrent transitions) [15], the dynamic aspect of a system and the rule R represents the system local transitions. Each rewrite rule in R has the form $(t \longrightarrow t')$ that can be expressed in terms of algebraic expressions (t and t') written in the Σ syntax [10] (see Section 2 in [10]). The rewrite theory $\mathcal{R} = (\Sigma, E, R)$, therefore, represents a concurrent system.

Several reported works [16, 17, 18, 19] present and discuss how to use rewriting logic and Maude to define and implement executable semantics for several languages, and how other semantic approaches are represented in rewriting logic [12]. Interested readers could refer to [10, 12, 20] for more details.

2.2 The Maude System

Maude is a high-performance implementation system supporting both *equational logic* and *rewriting logic* [21, 22, 23]. It is a formal tool environment that supports executable specifications, formal verification, declarative programming [22], parallel programming and rapid prototyping [24]. Maude can also be employed as a semantic framework to formally represent various systems such as distributed

algorithms, models of concurrency, the semantics of programming languages and network protocols[21]. Also, Maude feature rich, allowing for example specification of equational axioms such associativity and commutativity with identity.

A Maude module defines a precise mathematical model [25] and the essential unit of specification, which can be either a *functional module* or a *system module*. A functional module corresponds to a membership equational theory [26]. The following example illustrates a functional module in Maude that defines the natural number in *Peano* representation.

```
fmod NATURAL is
sort Nat .
op 0 : -> Nat [ctor] .
op s : Nat -> Nat [ctor] .
op _+_ : Nat Nat -> Nat [assoc comm id: 0] .
vars M N : Nat .
eq N + 0 = N .
eq N + s(M) = s(N + M) .
endfm
```

Here, a constant 0 and a successor operation `s` has been used to construct the data. Also, the addition operation is defined over the natural number and written in mixfix syntax and in a recursive way by structural induction on its second argument [27]. Table 2.1 describes the meaning of different Maude keywords. A system module defines a rewrite theory $R = (\Sigma, E \cup B, \mathrm{R}, \phi)$ and consists of sorts, equations, and rules. The following syntax shows the unconditional and conditional rules that are implemented in a system module.

$$\mathsf{rl}\ [\ \mathsf{l}\ ]\ :\ \mathsf{t}\ =>\ \mathsf{t}'\ .$$

$$\mathsf{crl}\ [\ \mathsf{l}\ ]\ :\ \mathsf{t}\ =>\ \mathsf{t}'\ \mathsf{if}\ \mathsf{C}\ .$$

The formal specification that implemented in Maude is executable. Therefore, there are several tools concerned with the analysis of Maude specifications such as the inductive theorem prover (ITP), the coherence checker tool (CRC), the Maude termination tool (MTT), etc. These tools are used to perform different types of formal analysis and verification such as verifying inductive properties of membership equational specification, simulation, and reachability analysis.

The *reduce* command, or the abbreviation *red* can be used to evaluate the equations in a functional module through the Maude interpreter, and the result printed with statistics about the execution [21]. Another Maude command that used to explore the behavior of system modules through the Maude interpreter is the *rewrite* command or the abbreviation *rew*. The rewrite command causes the specified term to be rewritten using the rules, equations, and membership axioms in the system module by using the top-down rule-fair strategy [21]. The *search* command is another Maude command used to explore the reachable state using breadth-first strategy. Also, the *search* command allows us to explore all possible solutions or impose a limit on the number of solutions because the number of such solutions could be infinite [21].

Table 2.1: Table of basic Maude keywords

Maude Keywords	Description
`sort(s)`	Used to represent data type or multiple data types. *Sorts* are popular term in an algebraic specification community. Sort syntax in Maude: sort $\langle$Sort$\rangle$. sorts $\langle$Sort $-$ 1$\rangle$ ... $\langle$Sort $-$ N$\rangle$.
`subsort(s)`	Used to define a subsort relation on sorts. For instance, $0 \in \mathbb{N}$. Subsort syntax in Maude: subsort $\langle$Sort $-$ 1$\rangle$ $<$ $\langle$Sort $-$ 2$\rangle$. subsorts $\langle$Sort $-$ 1$\rangle$ ... $\langle$Sort $-$ J$\rangle$ $<$ ... $<$ $\langle$Sort $-$ K$\rangle$ ... $\langle$Sort $-$ L$\rangle$.
`op(s)`	Used to declare Maude operator(s). The syntax in Maude: op $\langle$OpName$\rangle$: $\langle$Sort $-$ 1$\rangle$ ... $\langle$Sort $-$ K$\rangle$ $->$ $\langle$Sort$\rangle$ [$\langle$OpAttr$\rangle$] .
`var(s)`	Used to define a variable or variables in Maude. Variable syntax in Maude: var $\langle$varName$\rangle$: $\langle$Sort$\rangle$. vars $\langle$varName $-$ 1$\rangle$ ... $\langle$varName $-$ N$\rangle$: $\langle$Sort$\rangle$.
`eq`	Used to declare unconditional equations. Equation syntax in Maude: eq $\langle$Term $-$ 1$\rangle$ = $\langle$Term $-$ 2$\rangle$ [$\langle$EqAttr$\rangle$] .
`ceq`	Used to declare conditional equations. Conditional equation syntax in Maude: ceq $\langle$Term $-$ 1$\rangle$ = $\langle$Term $-$ 2$\rangle$if $\langle$EqCondition $-$ 1$\rangle$ /\ ... /\ $\langle$EqCondition $-$ N$\rangle$ [$\langle$CeqAttr$\rangle$] .
`rl`	Used to declare unconditional rewrite rules. Rewrite rule syntax in Maude: rl [$\langle$Label$\rangle$] : $\langle$Term $-$ 1$\rangle$ => $\langle$Term $-$ 2$\rangle$ [$\langle$RuleAttr$\rangle$] .
`crl`	Used to declare conditional rewrite rules. Conditional rewrite rule syntax in Maude: crl [$\langle$Label$\rangle$] : $\langle$Term $-$ 1$\rangle$ => $\langle$Term $-$ 2$\rangle$ if $\langle$EqCondition $-$ 1$\rangle$ /\ ... /\ $\langle$EqCondition $-$ N$\rangle$ [$\langle$CrlAttr$\rangle$] .

2.2.1 Inductive Theorem Prover (ITP Tool)

Recently, various inductive theorem provers have been developed such as the Maude ITP, Coq and HOL. Inductive Theorem Proving is one of the most successful verification approaches for proving complex properties of software algorithms.

The ITP tool [21, 27, 28, 29] is an experimental inductive theorem prover written entirely in Maude. It has been used to prove inductive properties of membership equational specification such as associativity, commutativity, etc.

The ITP tool implements a sound inference system for proving properties of Maude functional modules [27]. Therefore, the ITP tool has been used in this thesis to perform formal analysis and verification of the algebraic specification.

2.2.2 Reachability Analysis

Reachability is a fundamental problem that appears in several different contexts such as concurrent systems, finite-state machine and program analysis. In general, reachability problem is a decision on whether a certain computation system state is reachable based on its initial state and set of rules or transformation.

Maude is capable of performing *reachability analysis* by providing a variety of model checking and search commands to analyze and explore all possible behaviors using breadth-first strategy [21]. Maude allows for the specification of the number of rewrite steps, or satisfaction of other conditions which can be reached in a non-deterministic manner from the initial state [30].

The Maude *search* command has been used to perform *reachability analysis* by

finding states that are reachable from the initial state which matches the *search pattern* and satisfies the *search condition*. The search command has the following syntax (See Section 6.4.3 in [21] for more details).

search $\langle \mathsf{Term}-1 \rangle$ $\langle$ $\mathsf{SerachArrow}\rangle$ $\langle \mathsf{Term}-2 \rangle$ such that $\langle \mathsf{Conditon} \rangle$.

where,

- $\langle \mathsf{Term}-1 \rangle$ is the initial state (starting term).
- $\langle \mathsf{SerachArrow} \rangle$ is an arrow indicating the form of the rewriting proof from $\langle \mathsf{Term}-1 \rangle$ to $\langle \mathsf{Term}-2 \rangle$.
- $\langle \mathsf{Term}-2 \rangle$ is the pattern that has to be reached.
- $\langle \mathsf{Conditon} \rangle$ is an optional property that has to be satisfied by the reached state.

2.3 Overview of DReX

DReX [1] is a domain-specific language for string transformations. DReX is an extended work of [4] where Alur ***et al*** [4] proposed a group of combinators that capture regular string transformations. Moreover, these combinators are analogs to regular expression operations: concatenation ($R_1.R_2$), union ($R_1 + R_2$) and Kleene Closure (R^*) where R, R_1, and R_2 are regular expressions. However, the primary focus of [1] was the complexity of evaluating the output of a DReX program on a given input string by developing an efficient evaluation algorithm.

DReX has the following characteristics:

(i) Strong theoretical foundations. The class of functions expressible using DReX coincides with the class of regular string transformations. There are several equivalent characterizations, such as streaming string transducers, two-way finite state transducers, and graph transformations in monadic second-order logic.

(ii) Fast. There is a streaming one-pass algorithm to evaluate DReX programs.

(iii) Declarative. Transformations are modular, and small easy-to-understand transformers can be combined into more complicated ones.

(iv) Safe. It is possible to mechanically answer audit questions like, does this transformer ever emit an un-escaped backslash character? Equivalence and precondition computation are decidable.

The regular string transformation combinators that are supported by DReX are discussed in Table 2.2.

Table 2.2: DReX Combinators

DReX Combinator	Objective
`Base combinator`	$(\varphi \longmapsto \mathsf{d})(\sigma)$ The main objective of this combinator is to map any character "σ" that satisfies the predicate φ to string $d(\sigma)$.
`Split combinator`	$\mathsf{split}(\mathsf{f}, \mathsf{g})(\sigma)$ Unambiguously split string σ into two parts (σ_1, σ_2) then apply f over (σ_1) and g on (σ_2). The result will be the concatenation of the outputs that are obtained from f and g, respectively.
`Conditional combinator`	$(\mathsf{f}\ \mathsf{else}\ \mathsf{g})(\sigma)$ First, tries to apply $f(\sigma)$ if it is possible otherwise g will apply on σ.
`Combine combinator`	$\mathsf{combine}(\mathsf{f}, \mathsf{g})(\sigma)$ Is used to concatenate the output that is obtained from $f(\sigma)$ and $g(\sigma)$.
`Iterate combinator`	$\mathsf{iterate}(\mathsf{f})(\sigma)$ Unambiguously splits string σ into multiple parts $(\sigma_1\sigma_2...\sigma_n)$. Subsequently,$f$ will apply on each part. The final result will be the concatenation of the outputs that is obtained from evaluating f on each part.

2.3.1 Consistent DReX

Consistent DReX captures a restricted class of string transformations that assume unambiguous splits the input string (σ) into two or multiple parts using split and iterate combinator, respectively. The main purpose of this class is to provide an efficient evaluation algorithm by defining some consistency rules that restrict each operator. For instance, the split combinator can be used if the input string σ

can be uniquely divided into two parts $\sigma = \sigma_1\sigma_2$. So, in case the input string cannot be split, or if multiple viable splits exist, then the obtained result will be undefined ($\perp$).

Also, the consistency rules that have been used by consistent DReX match the concept of consistent unambiguous regular expressions (see Section 2.3 in [1]). These rules have been used to guarantee that the input string has a unique parse tree (σ does not have multiple viable splits).

The natural approach to evaluating string transformation program involves dynamic programming (DP) algorithms whereas the complexity of dynamic programming algorithm is cubic in size of the input string (σ) [1]. The DP algorithm has been used to handle all unrestricted DReX programs (see Section 4 in [1]) by evaluate each sub-program on each substring of the input that has cubic time complexity in the length of the input string, and does not scale to strings longer than approximately a thousand characters. The algorithm mimics the semantics of DReX by computing $out(f, \sigma, i, j)$ which represents the output of f on a portion of the input string σ $[\sigma[i, j]]$. The denotational semantic of consistent DReX proposed by Alur ***et al.*** in [1] can be found in Figure 2.1.

Moreover, the single pass algorithm has been used to evaluate consistent DReX program. This algorithm read the input string σ in a single left-to-right pass [1]. The complexity of this algorithm is linear in size of σ and polynomial in the size of the program.

$$OUT(bottom, \sigma, i, j) \ = \bot$$

$$OUT(\varphi \longmapsto d, \sigma, i, j) = \begin{cases} d(\sigma_i) & \text{if } i \ + \ 1 \ = \ j, \ and \ \varphi(\sigma_i) \ is \ true, \ and \\ \bot & otherwise. \end{cases}$$

$$OUT(\epsilon \longmapsto d, \sigma, i, j) = \begin{cases} d & \text{if } \sigma[i, j] \ = \ \epsilon, \ and \\ \bot & otherwise. \end{cases}$$

$$OUT(split(f_1, f_2), \sigma, i, j) = \begin{cases} \tau_1\tau_2 & \text{if } \exists! k \ such \ that \ i \ \leq \ k \ \leq \ j, \ and \ where \\ & \quad \tau_1 \ = \ OUT(f_1, \sigma, i, k) \ \neq \bot, \ and \\ & \quad \tau_2 \ = \ OUT(f_2, \sigma, k, j) \ \neq \bot, \ and \\ \bot & otherwise. \end{cases}$$

$$OUT(f_1 \ else \ f_2, \sigma, i, j) = \begin{cases} OUT(f_1, \sigma, i, j) & \text{if } OUT(f_1, \sigma, i, j) \ \neq \bot, \ and \\ OUT(f_2, \sigma, i, j) & otherwise. \end{cases}$$

$$OUT(combine(f_1, f_2), \sigma, i, j) \ = \ OUT(f_1, \sigma, i, j)OUT(f_2, \sigma, i, j)$$

$$OUT(iterate(f), \sigma, i, j) = \begin{cases} \epsilon & \text{if } i \ = \ j \ and \ OUT(f, \sigma, 0, 0) \ = \bot, \\ \tau_1\tau_2 & otherwise \ if \ \exists! k \ such \ that \ i \ \leq \ k \ \leq \ j, \ and \\ & \quad \tau_1 \ = \ OUT(iterate(f), \sigma, i, k) \ \neq \bot, \ and \\ & \quad \tau_2 \ = \ OUT(f, \sigma, k, j) \ \neq \bot, \ and \\ \bot & otherwise. \end{cases}$$

Figure 2.1: The denotational semantics of consistent DReX [1]

2.3.2 Inconsistent (Unrestricted) DReX

The consistent DReX is not the only way to represent regular string transformations. In fact, consistent DReX is a subset of regular string transformations that capture deterministic regular string transformations, for instance, the input string (σ) can be split uniquely into two parts ($\sigma \ = \sigma_1\sigma_2$). Moreover, the regular string transformations can be nondeterministic where the input string (σ) have a multiple valid split

CHAPTER 3

AN EQUATIONAL SPECIFICATION OF DREX

This chapter presents the core of our formal algebraic specifications implemented in Maude. We used the denotational semantics of DReX to develop algebraic semantics that captures *deterministic regular string transformations* using Maude. The proposed specification is executable which allows us to perform different types of formal analysis such as simulation and verify inductive properties of the equational specification.

The chapter is organized as follows. Section 3.1 discusses the abstract syntax and informal semantics of DReX where we discuss each syntactic category in details with some examples. In section 3.2, we present the algebraic semantics of *deterministic regular string transformations* followed by a detailed description of each equation. After that, the formal specification of deterministic regular string transformations implemented in Maude will be discussed in Section 3.3.

3.1 Abstract Syntax and Informal Semantics

DReX combinators describe how to capture and perform regular string transformations. The abstract syntax of DReX is shown in Figure 3.1 which consist of four syntactic categories and we will discuss each category in this section.

$$f \in Single - character\ function.$$

$$D \in \Sigma.$$

$$\sigma \in String.$$

$$\begin{aligned} e \in FunctionExrpession \ :== \ & f \mid \epsilon_d \mid \bot \\ \mid \ & split(e, e) \\ \mid \ & condition(e, e) \\ \mid \ & combine(e, e) \\ \mid \ & iterate(e, e) \end{aligned}$$

$$\begin{aligned} s \in StringExrpession \ :== \ & \sigma \mid \bot \\ \mid \ & apply(e, \sigma, i, j) \end{aligned}$$

Figure 3.1: Syntax of DReX

A single-character function, f, represents any regular function which is used perform a regular string transformations process. These functions map the input data strings to output data strings, where Σ describes the domain of these functions. Moreover, one character will be processed at a time where the *recursion*

method has been used to tackle the issue of $|\sigma| > 1$.

String, σ, represents the input parameter of a single character function where $\sigma \in \Sigma$.

Function Expression, e, can be one of the following seven expressions:

- f, represents any single character function that maps an input string into an output string ($\Sigma \longmapsto \Gamma$) where input and output strings are represented by Σ and Γ, respectively.

- ϵ_d, is a base combinator that maps empty string (ϵ) to the output d.

- $\perp$, is a base combinator that describes an undefined or error result for all input strings.

- Split combinator, $split(e, e)$, is a DReX combinator that takes two arguments. Each argument could be any *function expression* that belong to our abstract syntax. The split combinator is used to split unambiguously the input string σ into two parts (σ_1, σ_2), then the first expression will apply on σ_1 and the second expression will apply on σ_2. The final output is the concatenation of results obtained from the first and second expressions, respectively.

- Conditional combinator, $condition(e, e)$, is a DReX combinator that takes two arguments. Each argument could be any *function expression* that belongs to our abstract syntax. The first expression tries to evaluate the input string σ. If it fails, then the second expression will execute to evaluate σ.

- Combine combinator, $combine(e, e)$, is a DReX combinator that takes two arguments, each argument could be any *function expression* that belong to our abstract syntax. Combine combinator used to concatenate the two output strings that is obtained from applying each expression on σ.

- Iterate combinator, $iterate(f)$, is a DReX combinator that takes one arguments which can be any *function expression* that belongs to our abstract syntax. Iterate combinator used to split unambiguously the input string σ in multiple chunks ($\sigma_1\sigma_2...\sigma_n$) and outputs the concatenation of applying e on each of such chunk.

  Finally, **String expression,**s, can be either: (1) the input string (σ), (2) the error expression ($\perp$), which represents an error or undefined state when $\sigma \notin \Sigma$; or (3) the apply expression ($apply(e, \sigma, i, j)$); which represent the evaluation process of applying e on substring $\sigma[i, j]$. i *and* j are string indices, i is the begin index and j is the end index. This corresponds to the `OUT` function in the denotational semantics.

3.1.1 Small Examples

We now list a few examples of DReX expressions:

Table 3.1: Single-Character Function

RST Function (f)	Domain (Σ)	Codomain (Γ)	Description
`Caesar Encryption`	{"A","B","C","D"}	{"D","E","F","G"}	Single-character function that implements a Caesar Encryption technique.
`Caesar Decryption`	{"D","E","F","G"}	{"A","B","C","D"}	Single-character function that implements a Caesar Decryption technique.

Table 3.2: Function Expression

Function Expression (e)	Representation & Description
`condition(e,e)`	condition(Caesar Encryption, Caesar Decryption) Caesar Encryption will apply if it is possible otherwise Caesar Decryption will apply.
`iterate(e)`	iterate(UpperCase) Upper-Case function will apply multiple times.

Table 3.3: String Expression

String Expression (s)	Representation & Description
`apply(e,`σ`,i,j)`	`apply(split(Caesar Encryption, Caesar Decryption),"ABCDXY",1,4)` This syntax form will take the portion string (substring) from 1 to 4 ($\sigma[1,4]$) then it will find a unique split position where you can apply *Caesar Encryption* on the first part and then apply *Caesar Decryption* on the second part. After that, concatenates the obtained result.
`apply(e,`σ`,i,j)`	`apply(combine(Caesar Encryption, Caesar Decryption),"ABCZZAX",3,6)` This syntax form will take the substring ($\sigma[3,6]$) and apply *Caesar Encryption*, *Caesar Decryption* on $\sigma[3,6]$, respectively. Then, concatenates the obtained result.

3.2 Algebraic Semantics of DReX

An algebraic semantics is a formal method based on abstract algebra used to describe program semantics. In this thesis, algebraic semantics has been used to describe the *deterministic* regular string transformation process and perform formal analysis.

The equational theory that defines the algebraic semantics consists of signature and equation (Σ, E). Furthermore, to be able to define the algebraic semantics first we need to define sorts that represent data types. The main sorts in our specification are (i) string sort to represent string, (ii) function sort to represent regular string functions, and (iii) expression sort to represent the string combinators. Beside that, we define a set of operator that represent function. Each combinator in DReX maps to operator of sort combinator. Also, we are able to build a larger expression by combining those operator together. Moreover, we have a set of equations to present the behavior of our algebraic semantics of deterministic regular string transformations.

An algebraic semantics of DReX shown in Figure 3.2, the description of each equation can be found in Table 3.4 and the behavior of each combinator illustrated using flowchart in Figures 3.3-3.6. The whole definition is available in Appendix A.

$$apply(\bot, \sigma, i, j) \ = \bot \tag{BOTTOM}$$

$$apply(\epsilon_d, \sigma, i, j) = \begin{cases} d & \text{if } \sigma = \epsilon_d, \\ \bot & \text{otherwise.} \end{cases} \tag{EPSILON}$$

$$apply(f, \sigma, i, j) = \begin{cases} f(\sigma_{ij}) & \text{if } \sigma_{ij} = D(f) \\ \bot & \text{otherwise.} \end{cases} \tag{BASE}$$

$$apply(condition(e_1, e_2), \sigma, i, j) = \begin{cases} \bot & \text{if } \sigma_{ij} \notin D(condition(e_1, e_2)) \\ \sigma_1 & if \sigma_1 = apply(e_1, \sigma, i, j) \neq \bot \\ apply(e_2, \sigma, i, j) & \text{otherwise} \end{cases} \tag{CONDITIONAL}$$

$$apply(combine(e_1, e_2), \sigma, i, j) = \begin{cases} \sigma_1 + \sigma_2 & \text{if } \sigma_1 = apply(e_1, \sigma, i, j) \neq \bot \ and \\ & \quad \sigma_2 = apply(e_2, \sigma, i, j) \neq \bot, \\ \bot & otherwise \end{cases} \tag{COMBINE}$$

$$apply(iterate(e), \sigma, i, j) = \begin{cases} \bot & \text{if } \sigma_{ij} \notin D(iterate(e)) \ or \\ & \quad |splits(\sigma_{ij}, iterate(e), e)| \neq 1, \\ \sigma_1 + \sigma_2 & otherwise, where: \\ & \quad [k] = splits(\sigma_{ij}, iterate(e), e) \\ & \quad \sigma_1 = apply(iterate(e), \sigma, 0, k) \\ & \quad \sigma_2 = apply(e, \sigma, k, |\sigma_{ij}|) \end{cases} \tag{ITERATE}$$

$$apply(split(e_1, e_2), \sigma, i, j) = \begin{cases} \bot & \text{if } \sigma_{ij} \notin D(split(e_1, e_2)) \text{ or} \\ & |splits(\sigma_{ij}, e_1, e_2)| \neq 1, \\ \sigma_1 + \sigma_2 & otherwise, where: \\ & [k] = splits(\sigma_{ij}, e_1, e_2) \\ & \& \ \sigma_1 = apply(e_1, \sigma, 0, k) \\ & \& \ \sigma_2 = apply(e_1, \sigma, k, |\sigma_{ij}|) \end{cases} \tag{SPLIT}$$

$$splits(\sigma, e_1, e_2) = splitsPosition(\sigma, 0, e_1, e_2) \tag{SPLITS}$$

$$splitsPosition(\sigma, k, e_1, e_2) = \begin{cases} [] & \text{if } k > |\sigma| \\ L & \text{if } \sigma_1 \in D(e_1) \ \wedge \ \sigma_2 \in D(e_2), \\ & \text{and where:} \\ & \quad \sigma_1 = \sigma[0, k+1], \text{ and} \\ & \quad \sigma_2 \ = \ \sigma[k+1, |\sigma|], \text{ and} \\ & \quad L = [k] + + \ splitsPosition(\sigma, k+1, e_1, e_2) \\ splitsPosition(\sigma, k+1, e_1, e_2) & otherwise. \end{cases} \tag{splitsPosition}$$

Figure 3.2: Algebraic Semantics of DReX

Table 3.4: Algebraic Semantics of DReX description

Label	Description
BOTTOM	$\mathsf{apply}(\perp, \sigma, \mathsf{i}, \mathsf{j})$ Undefined or error for all input strings.
EPSILON	$\mathsf{apply}(\epsilon_{\mathsf{d}}, \sigma, \mathsf{i}, \mathsf{j})$ Map the input string σ_{ij} to constant d $[\epsilon \longrightarrow d]$ if the input string is ϵ otherwise undefined.
BASE	$\mathsf{apply}(\mathsf{f}, \sigma, \mathsf{i}, \mathsf{j})$ This semantic describe applying f on the portion string σ_{ij} iff the input string belong to the domain of f otherwise the obtained result will be error.
SPLIT	$\mathsf{apply}(\mathsf{split}(\mathsf{e}_1, \mathsf{e}_2), \sigma, \mathsf{i}, \mathsf{j})$ Here we have two cases that represent the obtained output. The first case describe the error in case of the portion input string $\notin$ domain of $split(e_1, e_2)$ or there are multiple split. The second case will apply if there exists a unique split $\sigma_{ij} = \sigma_1\sigma_2$ then concatenate the obtained result iff both e_1 and e_2 are defined.
CONDITIONAL	$\mathsf{apply}(\mathsf{condition}(\mathsf{e}_1, \mathsf{e}_2), \sigma, \mathsf{i}, \mathsf{j})$ The error will be the output in case of the input substring $\sigma_{ij} \notin$ domain $condition(e_1, e_2)$. If the first case doesn't match, then the first expression e_1 will apply on σij, and if this fails, applies e_2.
COMBINE	$\mathsf{apply}(\mathsf{combine}(\mathsf{e}_1, \mathsf{e}_2), \sigma, \mathsf{i}, \mathsf{j})$ Combine semantic is similar to the union operator of regular expression. If both $apply(e_1, \sigma, i, j)$ and $apply(e_2, \sigma, i, j)$ are defined then concatenate the obtained result from e_1 and e_2, respectively. However, if either expression is undefined for the input σ_{ij} then $apply(combine(e_1, e_2), \sigma, i, j)$ is undefined as well.
ITERATE	$\mathsf{apply}(\mathsf{iterate}(\mathsf{e}), \sigma, \mathsf{i}, \mathsf{j})$ This semantic is a counterpart of Kleene Closure-* of regular expression. If the input string $\sigma_{ij} \notin$ domain $iterate(e)$ or it can not be split or if multiple viable splits then $apply(iterate(e), \sigma, i, j)$ is undefined. Otherwise, $iterate(e)$ will apply on the input string σ_{ij}.

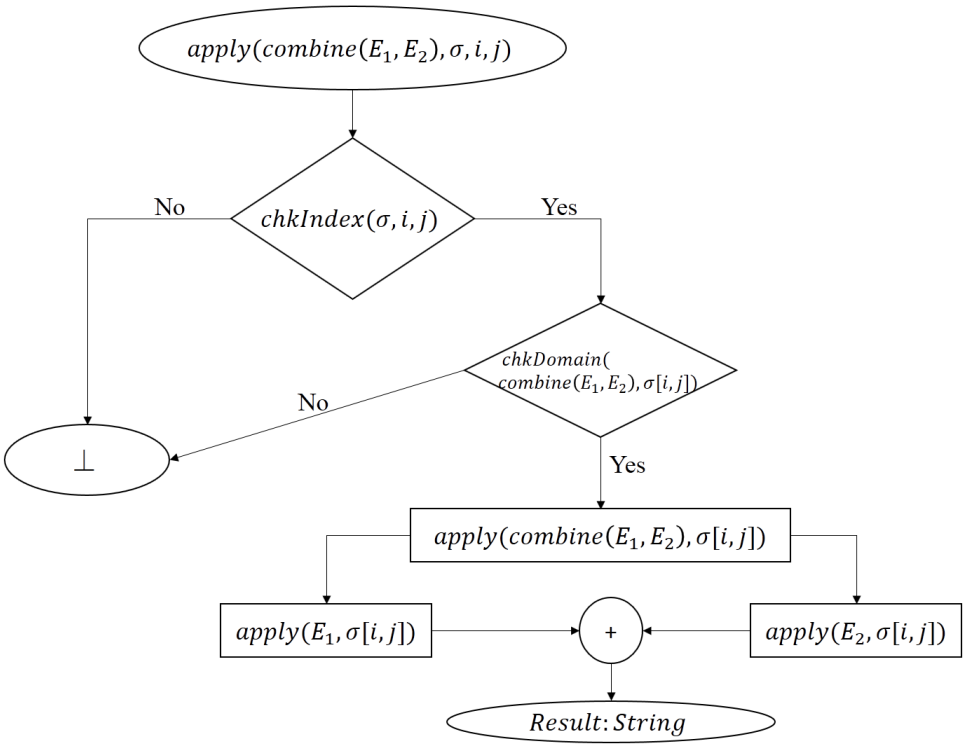

Figure 3.3: Combine Combinator Flowchart

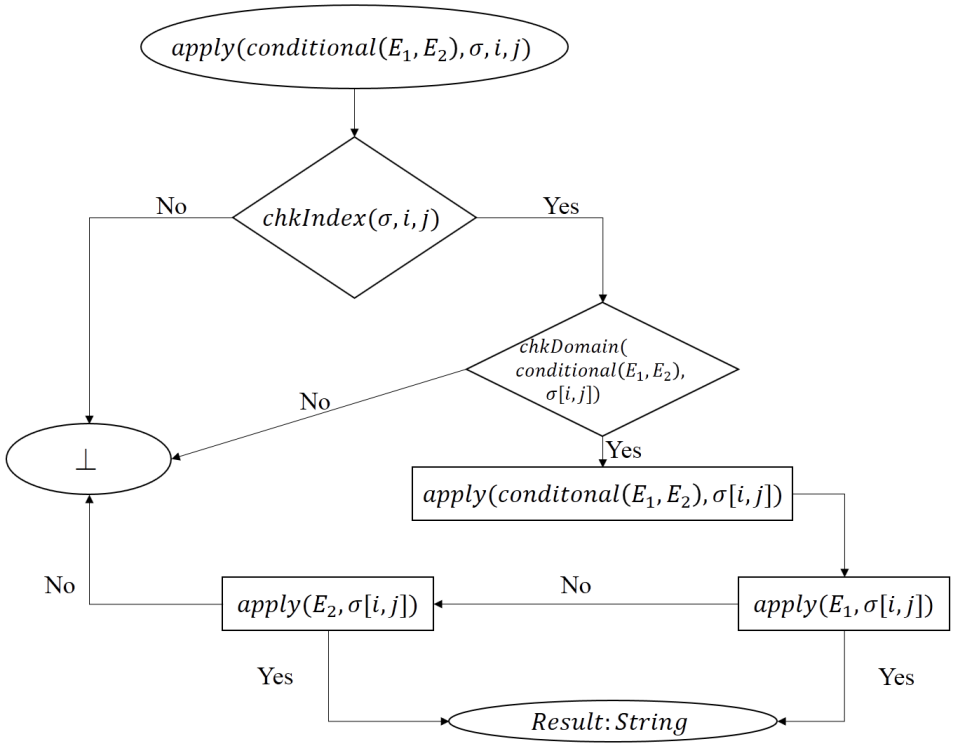

Figure 3.4: Conditional Combinator Flowchart

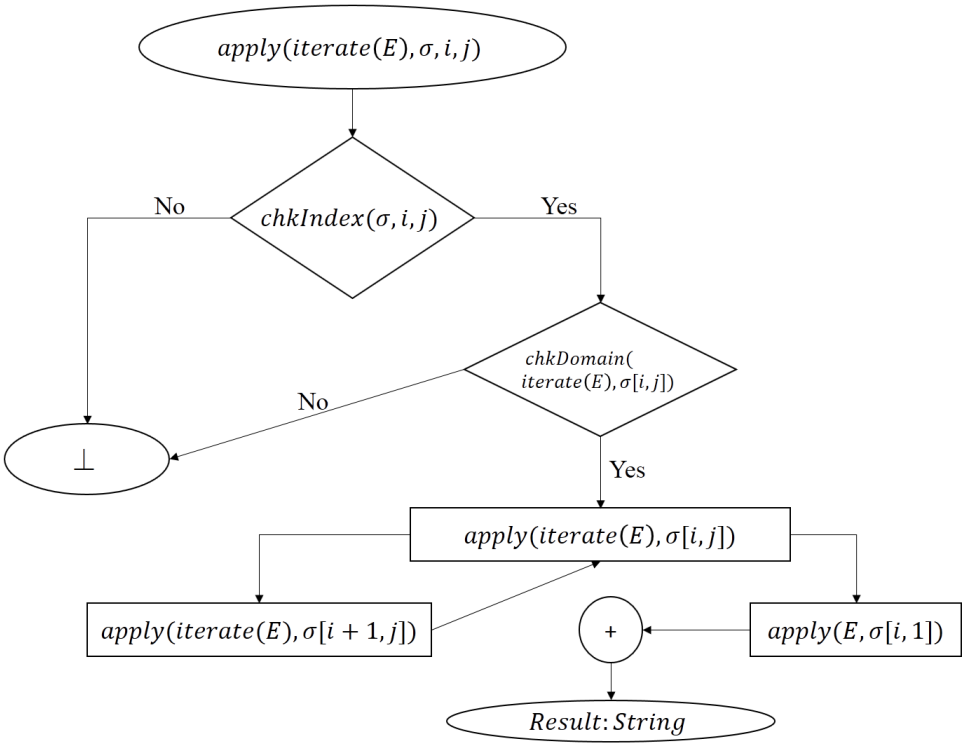

Figure 3.5: Iterate Combinator Flowchart

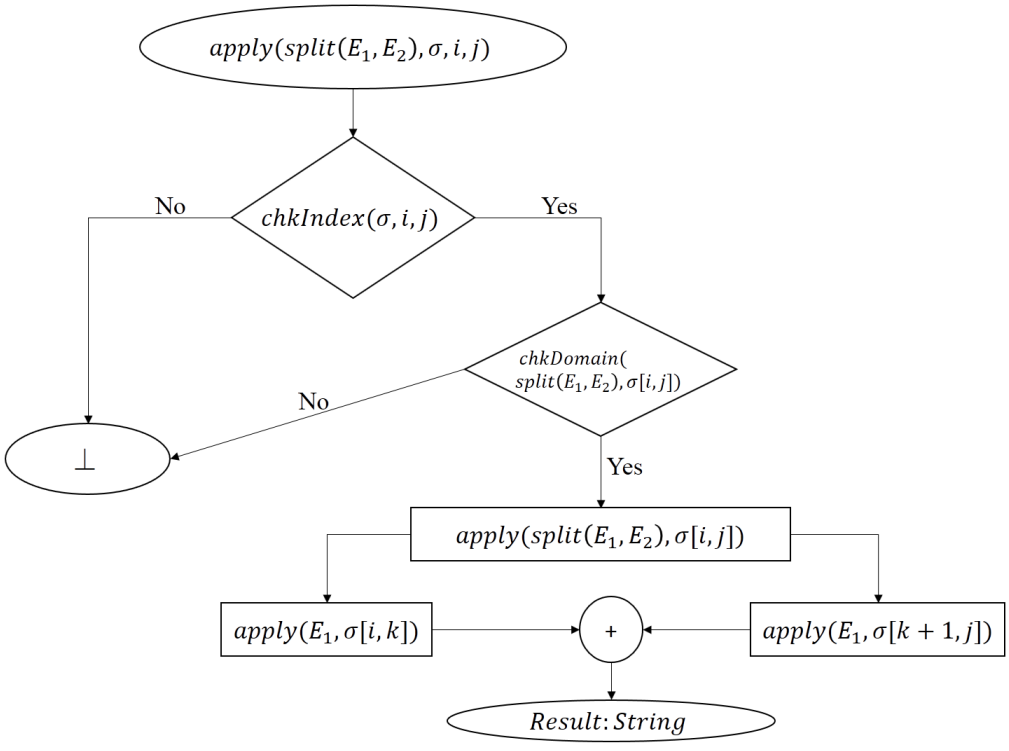

Figure 3.6: Split Combinator Flowchart

3.3 Formal Specification of Deterministic Regular String Transformations in Maude

In our research, Maude has been used to develop our framework due its ability to produce executable specification for diverse logics, theorem provers and models of computation [31]. Also, Maude has previously been effectively used in numerous applications and in software engineering tools [32].

3.3.1 Regular Expressions in Maude

Language is a set of string and we utilize regular expression in this work due its ability to describe languages and it is an algebraic way to define patterns.

In our research, we utilize regular expressions by adapting Sen and Rosu's algebraic specification of extended regular expressions (ERE's) with a simplified axiomatization of regular expressions using subsorting [33] to simulate regular string transformations combinator behaviors as proposed by Alur ***et al.*** in [1]. In addition, regular expression has been used to define expression e domains and verify membership of input strings in them [$\sigma \in D(e)$]. The following RE module represents regular expression operations (.,+,*) using Maude.

```
fmod RE is
   *** NeeRe: A regular expression that
   *** is neither empty nor epsilon
   sorts Event NeeRe Re .
```

```
    subsort Event < NeeRe < Re .

    *** the singlton language containing only epsilon
    op epsilon : -> Re .

    *** the empty language
    op empty : -> Re .

    *** Alternation (the union of two regular languages)
    op _+_ : Re    Re -> Re    [assoc comm id: empty prec 60] .
    op _+_ : NeeRe Re -> NeeRe [ditto] .

    *** Concatenation
    op __  : Re    Re    -> Re [assoc id: epsilon prec 50] .
    op __  : NeeRe NeeRe -> NeeRe [ditto] .

    *** Kleene star
    op (_*) : Re    -> Re    [prec 40] .
    op (_*) : NeeRe -> NeeRe [ditto] .
endfm
```

The above Maude specification illustrates the regular expression (RE) module. Three sorts (types) have been defined to represent RE (Event, Re, NeeRe). The

RE's define languages inductively by applying concatenation (.), union (+) and Kleene Closure (*). To capture these operations, we define eight operations to describe RE operations as shows in RE module. Also, the result of applying RE operations will be RE.

Therefore, for an alphabet (Σ), RE over Σ is defined as follows: $\mathsf{RE} ::= \epsilon \mid \phi \mid \mathsf{Re\ Re} \mid \mathsf{NeeRe\ NeeRe} \mid \mathsf{Re} + \mathsf{Re} \mid \mathsf{NeeRe} + \mathsf{NeeRe} \mid (\mathsf{Re})^* \mid (\mathsf{NeeRe})^*$.

The *ditto* attribute can be given to an operator for which another subsort-overloaded instance has already appeared, either in the same module or in a submodule. Furthermore, it is forbidden to use ditto on the first declared instance of an operator, since this is nonsensical [21].

3.3.2 From String to Regular Expression

In this research, we focus on regular string transformations where the domain of f or e is a string. Therefore, in order to verify the input string ($\sigma \in Domain(e)$) we are converting the string domain to a regular expression. We demonstrate how to convert the string (function domain) to a regular expressions using the following operator in Maude:

$$op\ ev\ :\ String\ \sim>\ Re\ .$$

$$op\ ev\ :\ Char\ ->\ Re\ .$$

Moreover, the following example illustrates how to convert string domain (Σ) to a regular expression using Maude syntax. For instance, let assume that

we have a regular string transformations function f that defined over the set {"A","B"} ($\mathsf{f} : \Sigma = \{\text{"A"}, \text{"B"}\}$), the new form of Σ represented in Maude is $\Sigma = \{\mathsf{ev}(\text{"A"})\mathsf{ev}(\text{"B"})\}$.

3.3.3 Membership Verification in Maude

In this section, we will demonstrate how to verify the input string (σ) to ensure it is valid ($\sigma \in Domain$) using a membership operator implemented in Maude. The membership operator will return *true* or *false* for the valid and invalid input string (σ), respectively. The following Maude specification show the representation of membership operator.

```
--- Membership operation
op _in_ : Re Re -> Bool [prec 80] .
```

The membership operator is defined inductively on the structure of a regular expression, which is constructed using concatenation, union and Kleene-* operations of regular expression. The algebraic specification that used to solve the membership problem for regular expressions represented in Maude syntax can be found in Appendix A.

This algebraic specification consists of a set of equations and conditional equations which used to verify the membership in the regular expression, a heavy adaptation of Sen and Rosu's algebraic specification [33]. The Interested reader could refer to [33] for more details.

3.3.4 Regular String Function Representation

In this section, we will show how to define a regular string function using Maude syntax based on our specification where the regular string function consist of (i) name, (ii) domain, and (iii) behavior. The following syntax form show how to specify regular string function f in Maude based on our specification.

```
--- A function as a pair of a name and a domain
op (_,_) : String Re -> Function .
--- The domain of a function
op domain : Function -> Re .
 --- An interface for a function's behavior
op applyFunction : String String -> String .
```

The following example shows a single character function that developed in our specification to represent a *Caesar Encryption*. In this particular example, $\Sigma = \{"A","B","C","D"\}$ (function domain) and the *conditional equation (applyFunction)* specify the formal behavior of *Caesar Encryption* .

```
op  CaesarEncryption : -> Function .
eq  CaesarEncryption = ("CaesarEncryption", (ev("A") + ev ("B")
                                          + ev("C") + ev("D")))  .
ceq applyFunction("CaesarEncryption",S) =
        if (Len == 0) then
                   ""
        else( if Len == 1 then
                   char(ascii(S) + 3)
              else
                   char(ascii(substr(S,0,1)) + 3) +
                   applyFunction("CaesarEncryption",substr(S,1,Len))
                   fi) fi if Len := length(S) .
```

3.3.5 Split Position Algorithm

The *split and iterate combinators* proposed in [1] need to verify and ensure that the input string could be unambiguously split into two parts in case of the split combinator or into multiple parts in order to use the *iterate combinator*.

In order to use these two combinators in our specification we proposed *split position algorithm* 1 to determine valid split positions. This algorithm will return a list of valid indices if they exists.

Consistent DReX [1] deals with a list that have one item (position) to apply the *split and iterate combinators* (unique split) otherwise DReX failed to evaluate the input string. However, we are able to tackle this constraint by extending DReX where we developed NDREX using rewriting rule to capture non-deterministic regular string transformation where the *split position algorithm* 1 return a list that contains more than one valid index.

The algebraic specification of the split position algorithm in Maude could be found in Appendix A.

The main objective of this algorithm is to find all indices representing valid splits, where f and g represent regular string functions, the input string represented using σ, i and j represent string indices, and ks represent a list that contains all valid indices. First, we verify i and j over $sigma$ using $chkIndex$ function that returns either $true$ or $false$, in the case of $false$ we are not able to use this algorithm an we end up with an error. Otherwise, we will find all the valid indices using a loop statement wherein each iteration we split the input

Algorithm 1: Split Position algorithm for finding all indices representing valid splits

Input : $f,\ g,\ \sigma,\ i,\ j$
Output: The list of all indices representing valid splits or error

```
1  if checkindex(σ, i, j) = false then
2  |  error
3  else
4  |  Let k ⟵ i ; ks ⟵ []
   |   while (k ≤ j) do
5  |  |  Let s1 ⟵ substring(σ, i, k)
6  |  |  Let s2 ⟵ substring(σ, k + 1, j)
7  |  |  if s1 ∈ domain(f) and s2 ∈ domain(g)
8  |  |       ks ⟵ ks ++ [k]
9  |  |  k ⟵ k + 1
10 |  end
11 |  return ks
12 end
```

string σ into two parts s_1 and s_2 and verify $s_1 \in domain(f)$ and $s_2 \in domain(g)$, if the two conditions are correct we add this valid position represented in k to the list, otherwise we move the k to next position and repeat the process until $k \leq j$.

3.3.6 Deterministic Regular String Transformation Modules

Maude has been used to develop deterministic regular string transformations modules. There are 14 modules implemented in our specification to capture deterministic regular string transformations based on the denotational semantics of DReX [1].

Table 3.5 shows the implemented modules that has been used in our specification along with a brief description of each. The Maude definition of each module could be found in Appendix A.

Once the definition is complete and saved in a ".maude", say drex.maude, the next step is to compile the desired file. This is done with the **load** command:

```
load drex.maude .
```

Table 3.5: Deterministic Regular String Transformation Modules in Maude.

Module Name	Description
Core modules	The core semantics modules specify the semantics of regular string transformation. Each module represents different task.
`RE`	An adaptation of Sen and Rosu's algebraic specification [33] of Extended Regular Expression's (ERE's) with a simplified axiomatization of regular expressions (RE) using subsorting.
`SYMPLIFY_RE`	Equational simplification of RE, a heavy adaptation of Sen and Rosu's specification [33].
`RE_MEMBERSHIP`	Membership in RE's, a heavy adaptation of Sen and Rosu's specification [33].
`POS-LIST`	Provides a list that contains all possible split positions.
`FUNCTION`	An abstract representation of a regular transformation function.
`FUNCTION_AUX`	Auxiliary operators used to (i) converts a string to a regular expression, (ii) check membership of a string in the domain of a function, and (iii) check that the indices I and J are valid w.r.t. the given string.
`DREX-SYNTAX`	Represent the syntax of DReX: (i) base combinators, and (ii) main combinators.
`SEM-INTERFACE`	The semantics interface that need to be defined for all combinators.
`BASE-SEMANTICS`	Semantics of the base combinators.
`COMBINE-SEMANTICS`	Semantics of the combine combinator to evaluate $apply(cobmine(e,e),\sigma,i,j)$
`COND-SEMANTICS`	Semantics of the conditional combinator to evaluate $apply(condition(e,e),\sigma,i,j)$
`ITER-SEMANTICS`	Semantics of the iterate combinator to evaluate $apply(iterate(e),\sigma,i,j)$
`SPLIT-SEMANTICS`	Semantics of the split combinator to evaluate $apply(split(e,e),\sigma,i,j)$
`DREX-SEMANTICS`	Represent an interface to access our regular string transformation specifications.

3.3.7 Algebraic Specification of Combine Combinator

In this part, we are discussing the algebraic specification of *combine combinator* in Maude. As we mentioned earlier, this combinator used to concatenate the

obtained result from $f(\sigma)$ and $g(\sigma)$. The following functional module represents an algebraic specification of the combine combinator in Maude.

```
fmod COMBINE-SEMANTICS is
inc SEM-INTERFACE .

vars I J       : Nat .
vars E1 E2     : Expr .
vars S SE1 SE2 : String .

*** The domain of combine (assumes consistent DReX!)
ceq domain(combine(E1,E2))
= domain(E1) if domain(E1) == domain(E2) .

*** Semantics of combine
ceq apply(combine(E1,E2),S,I,J) = SE1 + SE2
if domain(combine(E1,E2)) :: Re
/\ SE1 := apply(E1,S,I,J)
/\ SE2 := apply(E2,S,I,J) .
endfm
```

We are importing a module `SEM-INTERFACE` into `COMBINE-SEMANTICS` using *inc* (include) Maude keyword, declare seven variables of types `Nat` (Natural Number), `Expr` (Function-Expression), and string using *vars* (variables) Maude keyword. Also, we define two conditional equations using *ceq* Maude keyword to capture the combine combinator behavior. The first conditional equation represents the domain of combine. The second one describes the combine combinator behavior which verifies the domain by reducing the $combine(E1, E2)$ to its canonical form and checking that it is equal to Re using membership operator (::), then we use the *matching equations* of the form $(t := t')$ in this conditional equation to match the obtained result of evaluating $E_1(\sigma_{ij})$ and $E_2(\sigma_{ij})$ to the string.

3.3.8 Algebraic Specification of Split Combinator

Another regular string transformations implemented algebraically in Maude which used to evaluate $e_1(\sigma_1)$ and $e_2(\sigma_2)$ if there exists a unique split $\sigma_{ij} = \sigma_1\sigma_2$ as discussed earlier is the split combinator. The following `SPLIT-SEMANTICS` module (functional module) represents an algebraic specification of the split combinator in Maude.

```
fmod SPLIT-SEMANTICS is
inc SEM-INTERFACE .
pr POS-LIST .
vars S S1 S2 S'  : String .
vars E1 E2       : Expr .
vars BinD1 BinD2 : Bool .
vars K I J       : Nat .
vars L1 L2       : List .

*** Domain of a split expression
eq domain(split(E1,E2)) = domain(E1) domain(E2) .

*** Semantics of split
ceq apply(split(E1,E2),S,I,J) = S1 + S2
if chkIndex(S,I,J)
/\ S' := substr(S,I, sd(J, I))
/\ K  := splits(S',E1,E2)
/\ S1 := apply(E1, S', 0, K + 1)
/\ S2 := apply(E2, S', K + 1, length(S')) .

*** The split position(s) in
*** a string given two expressions
op splits : String Expr Expr -> Nat .
eq splits(S, E1, E2) = splitsPosition(S, 0, E1, E2) .

*** a helper function for the
*** split position(s) algorithm
```

```
op splitsPosition : String Nat Expr Expr -> List .
ceq splitsPosition(S,K,E1,E2) = mt if K > length(S) .
ceq splitsPosition(S,K,E1,E2)
= if BinD1 and BinD2 then
K, splitsPosition(S, K + 1,E1, E2)
else
splitsPosition(S, K + 1, E1, E2) fi
if K <= length(S)
/\ BinD1 := chkDomain(E1,substr(S,0,K + 1))
/\ BinD2 := chkDomain(E2,substr(S,K + 1,length(S))) .
endfm
```

We are importing SEM-INTERFACE and POS-LIST into COMBINE-SEMANTICS using *inc* (include) and *pr* (protecting) Maude keywords. Declaring a set of variables of type String, Expr (Function-Expression), Bool (boolean), Nat (Natural Number), and List represents the correct split positions. The domain of this combinator is the concatenation of both $D(e_1)$ and $D(e_2)$ that presented algebraically by defining an equation using *eq* Maude keyword. The semantics of split combinator implemented using conditional equation. In this equation, we are using the *matching equations* to evaluate the right-hand side expression and reduce to its canonical form and checking that matches the left-hand side term. Another two conditional equations splits and splitsPosition used in the split semantics as a helper functions to find valid split positions and return it as a list of natural number. In this specification, if the list contains more than one valid position the matching equation will fail because of k defined as Nat which can't handle multi-values.

3.3.9 Validation of The Semantics

In this section, we checked the validation of our semantics through a test suite using a bunch examples. Theses cases are small, straightforward and their objective is to figure out and identify the semantics problems. Moreover, these examples are designed in a way that each of them essentially cover a certain aspect of DReX. Hence, this test suite allowed us to perform a proper modification to fix these bugs and were run using Maude interpreter.

In the beginning, we observed unexpected behavior from the specification which indicates that our specification required more revision and investigation to a obtain correct result. Therefore, we performed the required modification and all these examples passed this test.

CHAPTER 4

FORMAL ANALYSIS OF DREX EXPRESSIONS

In this chapter, we show the results obtained through the execution of revised algebraic semantics of DReX on different sample programs of *deterministic* regular string transformations using Maude rewriting logic engine. In addition, we did formal analysis to verify and emphasize the correctness of our proposed specification using the inductive theorem prover tool.

4.1 Simple Expression Examples

In this section, we show the result of executing single-character and function expressions using Maude interpreter based on the algebraic specifications of regular string transformations.

4.1.1 Single Character Function Expression

Example 4.1.1.1 The following example shows the result of executing *Caesar Encryption* function. The string transformation will be ["BC" $\longmapsto$ "EF"].

```
Maude> red apply(CaesarEncryption,"ABCDEFG",1,3) .
reduce in FUNCTION-LIB : apply(CaesarEncryption, "
   ABCDEFG", 1, 3) .
rewrites: 100 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "EF"
```

Example 4.1.1.2 This example shows the result of executing *Caesar Decryption* function. In this case, the string transformation will be ["XG" $\longmapsto \perp$] due to invalid input character $X \notin \Sigma$. The algebraic specification tried to evaluate the input string (σ) and reduce it to the canonical form. However, since the input string (σ) does not belong to the domain of the function, the algebraic specification failed to reduce it and produced an error. The error is represented by a term that does not have a sort but is of the kind (`[String]`) as shown in the following example.

```
Maude>  red apply(CaesarDecryption,"ABXGRL",2,4) .
reduce in FUNCTION-LIB : apply(CaesarDecryption, "ABXGRL
   ", 2, 4) .
rewrites: 57 in 0ms cpu (0ms real) (~ rewrites/second)
 result [String]: apply("CaesarDecryption",(ev("D") + ev
   ("E") + ev("F") + ev(
```

```
"G")) *, "ABXGRL", 2, 4)
```

Example 4.1.1.3 Another example of single character function that convert lower case character to upper case [”a” $\longmapsto$ ”A”] would be evaluated by executing the *Upper Case* function.

```
Maude> red apply(UpperCase,"a",0,1) .

reduce in FUNCTION-LIB : apply(UpperCase, "a", 0, 1) .

rewrites: 58 in 0ms cpu (0ms real) (~ rewrites/second)

result Char: "A"
```

Example 4.1.1.4 In this example, we will execute the *lower Case2* function to convert the upper case characters to lower case. The transformation result is [”BC” $\longmapsto$ $\perp$] because the input character ($\sigma =$ ”BC”) is invalid $\sigma \notin \Sigma$.

```
Maude>  red apply(LowerCase2,"AaBCDBBXR",2,4) .

reduce in FUNCTION-LIB : apply(LowerCase2, "AaBCDBBXR",
   2, 4) .

rewrites: 56 in 0ms cpu (0ms real) (~ rewrites/second)

result [String]: apply("ConvertToLower2",(ev("A") + ev("
   B") + ev("C") + ev(

"D")), "AaBCDBBXR", 2, 4)
```

4.1.2 Function Expression Examples

Example 4.1.2.1 In this example, the *combine combinator* would be used to apply both the *Caesar Encryption* and the *Lower Case* functions on a portion of the input string and subsequently concatenate the obtained results.

```
Maude> red apply(combine(CaesarEncryption,LowerCase),"
   ABCByz",1,4) .
reduce in FUNCTION-LIB : apply(combine(CaesarEncryption,
    LowerCase), "ABCByz",
1, 4) .
rewrites: 286 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "EFEbcb"
```

Example 4.1.2.2 In this case, the *condition combinator* would try to apply *Caesar Decryption* first, if the obtained result gives $\perp$ then the *condition combinator* will execute the *Lower Case* and display the obtained result.

```
Maude>  red apply(conditional(CaesarDecryption,LowerCase
   ),"ABCByz",1,4) .
reduce in FUNCTION-LIB : apply(conditional(
   CaesarDecryption, LowerCase),
"ABCByz", 1, 4) .
rewrites: 301 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "bcb"
```

Example 4.1.2.3 In this case, the *condition combinator* produce an error ($\perp$) after checking the indices value. This happens because our designed framework detected that the j index value is out of bound.

```
Maude> red apply(conditional(CaesarDecryption,LowerCase)
   ,"ABCByz",0,8) .
reduce in FUNCTION-LIB : apply(conditional(
   CaesarDecryption, LowerCase),
"ABCByz", 0, 8) .
rewrites: 66 in 0ms cpu (0ms real) (~ rewrites/second)
result [String]: apply(conditional("CaesarDecryption",(
   ev("D") + ev("E") + ev("F") + ev("G")) *, "
   ConvertToLower",(ev("A") + ev("B") + ev("C") + ev("D"
   ))*), "ABCByz", 0, 8)
```

The algebraic specification tried to evaluate the input string (σ) and reduce it to the canonical form. However, since the input string (σ) does not belong to the domain of the function, the algebraic specification failed and produced an error. The error is represented by a term that does not have a sort but is of the kind (`[String]`) as shown in the above example.

Example 4.1.2.4 The obtained result in example 4.1.4 was $\perp$ because the input string does not belong to the domain of the *Lower Case2*. In this example, we will use the *iterate combinator* with the *Lower Case2 function* which converts the domain from [$\Sigma \longmapsto \Sigma^*$]. So, the transformation result is [BC $\longmapsto$ bc].

```
Maude> red apply(iterate(LowerCase),"AaBCDBBXR",2,4) .
reduce in FUNCTION-LIB : apply(iterate(LowerCase), "
   AaBCDBBXR", 2, 4) .
rewrites: 185 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "bc"
```

Example 4.1.2.5 In this example, the *split combinator* would be used to execute the *Caesar Encryption* and the *Caesar Decryption* functions on substring (”ABF”). The input string can be **split uniquely** into two parts where $\sigma_1 =$ ”AB” and $\sigma_2 =$ ”F”. The obtained result is a concatenation of executing the *Caesar Encryption* on σ_1 and executing the *Caesar Decryption* on σ_2.

```
Maude> red apply(split(CaesarEncryption,CaesarDecryption
   ),"ABFG",0,3) .
reduce in FUNCTION-LIB : apply(split(CaesarEncryption,
   CaesarDecryption),
"ABFG", 0, 3) .
rewrites: 562 in 4ms cpu (1ms real) (140500 rewrites/
   second)
result String: "DEC"
```

Example 4.1.2.6 In this example, the obtained result of applying the *split combinator* is $\perp$ because the input string can be *split into multiple chunks*. For instance, $\sigma_1 =$ ”B” and $\sigma_2 =$ ”CD”, or $\sigma_1 =$ ”BC” and $\sigma_2 =$ ”D”.

```
Maude> red apply(split(CaesarEncryption,LowerCase),"
   ABCDCCC",1,4) .
reduce in FUNCTION-LIB : apply(split(CaesarEncryption,
   LowerCase), "ABCDCCC",
1, 4) .
rewrites: 397 in 0ms cpu (0ms real) (~ rewrites/second)
result [String]: apply(split("CaesarEncryption",(ev("A")
    + ev("B") + ev("C") +
ev("D")) *, "ConvertToLower",(ev("A") + ev("B") + ev("C"
   ) + ev("D")) *),
"ABCDCCC", 1, 4)
```

4.2 Larger Expression Examples

Here, we show the result of executing nested expressions and a case study inspired by [1] using Maude interpreter based on the algebraic specifications of regular string transformations.

4.2.1 Nested Expression Examples

In the following examples, we used nested combinator to verify and check our semantics. Each example describe different nested combinators and show the obtained results.

Example 4.2.1.1 This example illustrates the obtained result of applying the *nested conditional combinator* over the input string ($\sigma = "ABC"$). The first argument fails to evaluate σ. Therefore, the second argument has been used to evaluate σ and display the output.

```
Maude> red apply(conditional(combine(CaesarEncryption,
   CaesarDecryption),combine(LowerCase,LowerCase)),"ABCD
   ",0,3) .
reduce in FUNCTION-LIB : apply(conditional(combine(
   CaesarEncryption,
CaesarDecryption), combine(LowerCase, LowerCase)), "ABCD
   ", 0, 3) .
rewrites: 308 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "abcabc"
```

Example 4.2.1.2 This example illustrates the error ($\perp$) case. First and second argument try to evaluate the input string because the *nested conditional combinator* has been used. However, both of them fails and as a result $\perp$ output was obtained.

```
Maude> red apply(conditional(combine(CaesarEncryption,
   CaesarDecryption),combine(LowerCase,UpperCase)),"
   ADDFD",1,2) .
reduce in FUNCTION-LIB : apply(conditional(combine(
   CaesarEncryption,
```

```
CaesarDecryption), combine(LowerCase, UpperCase)), "
   ADDFD", 1, 2) .
rewrites: 51 in 0ms cpu (0ms real) (~ rewrites/second)
result [String]: apply(conditional(combine("
   CaesarEncryption",(ev("A") + ev(
"B") + ev("C") + ev("D")) *, "CaesarDecryption",(ev("D")
    + ev("E") + ev(
"F") + ev("G")) *), combine("ConvertToLower",(ev("A") +
   ev("B") + ev("C") +
ev("D")) *, "ConvertToUpper",(ev("a") + ev("b") + ev("c"
   ) + ev("d")) *)),
"ADDFD", 1, 2)
```

The algebraic specification tried to evaluate the input string (σ) and reduce it to the canonical form. However, since the string of entry (σ) does not belong to the domain of the function, the algebraic specification failed and produced an error. The error is represented by a term that does not have a sort but is of the kind (`[String]`) as shown in the above example.

4.2.2 Delete One-Line Comments

This case study inspired by [1], different type of regular string functions and combinators have been used to deletes all one-line comments from a particular example. The comment line starting with // where the program can be encoded

using `C, Java or C#`.

The input string will be either a comment or non-comment line of the form ($//\sigma\backslash n$) or ($\sigma\backslash n$), respectively. The program reads a line and deletes it if it is a comment line and copies it otherwise.

This program consists of 5 modules implemented in Maude based on our algebraic specification. Table 4.1 shows the implemented modules used in our specification along with a brief description of each. The Maude definition of each module could be found in Appendix A.

Table 4.1: Delete one-line comments Modules in Maude.

Module Name	Description
`DEL-FUNCTION`	Contain regular string functions that have been used to delete single character.
`COPY-FUNCTION`	Consist of different regular functions that used to copy single character.
`DEL-COMBINATORS`	DReX combinators have been used to delete comment line using DEL-FUNCTION module .
`COPY-COMBINATORS`	DReX combinators have been used to copy non-comment line using COPY-FUNCTION module .
`CODE-TRANSFORMATION`	Invoke DEL-COMBINATOR and COPY-COMBINATOR to process the input string.

The following particular examples in Table 4.2.2 illustrate delete one-line comment for different cases.

Table 4.2: Delete One-Line Comment Examples

The Input String (σ)	Obtained Result	Description
//Comment line	ϵ	This code contain only one line comment. Therefore, the comment line deleted and the obtain result is ϵ (empty string)
int x = 10 ;	int x = 10 ;	Here, we have only one line that declare integer variable, so the program will copy the input string and retrieve it as it is.

//Declare integer variables
int x, y, z ;
//Define double variable
double a = 25.5;
//Additon equation
x = (y + z) + a;
	int x, y, z ;
	double a = 25.5;
	x = (y + z) + a;
		In this particular example, we have a combination of comment and non-comment lines. The program reads a line and copies it if it isn't comment line, otherwise deletes it.

The following Maude result of evaluating the third example that shown in Table 4.2.2 is described bellow. The `delete-comm` equation received the input string (σ) that presented using `sampleCode1` variable and reduce to canonical form, each line end with $\backslash n$. The program reads a line and copies it if it isn't comment line, otherwise deletes it.

```
Maude> red delete-comm(sampleCode1) .

reduce in EVAL : delete-comm(sampleCode1) .

rewrites: 2446676 in 2488ms cpu (3413ms real) (983390
   rewrites/second)
result String: "int␣x,y,z␣;\n␣double␣a␣=␣25.5;\n␣x␣=␣(y␣
```

```
+␣z)␣+␣a;"
```

4.3 Formal Analysis using the ITP

Once the system specifications developed in Maude, we obtained an algebraic specification of the system. As these specifications are executable, they can be used to analyze and simulate the system. In this section, we will use Maude's inductive theorem prover (ITP) to verify inductive properties of membership equational specification of functional modules such as idempotency and associativity of the deterministic regular string transformations.

- **Proving Idempotency Property of the Conditional Combinator**

  Idempotence is the property of particular function or operation where the result or effect of executing it multiple times for a given input is the same as executing it only once for the same input. Therefore, if we know that an operation is idempotent, we can run it as many times as we like. The mathematical presentation of idempotency property is $\mathsf{f}(\mathsf{x},\mathsf{x}) = \mathsf{x}\ (\forall \mathsf{x})$.

  Consequently, we will use the ITP tool to prove the idemoptency property of the *conditional combinator*. To do that, the actual goal we want to prove needs to be loaded where the syntax for entering goal is: (goal goalname : modulname | − formula .) [29] after that the user will use **ind** command to discharge the goal where **ind** command *"takes a variable x*

that is universally quantified in the current formula, and perform structural induction on x using the corresponding syntax ind on x .*)"* [29].

Symbolic Proof using ITP. For a Maude module representing the conditional combinator (COND-SEMANTICS), we would like to show that the conditional combinator is idempotent, where $\mathsf{apply}(\mathsf{conditional}(\mathsf{E},\mathsf{E},\sigma,\mathsf{I},\mathsf{J})) \equiv \mathsf{apply}(\mathsf{E},\sigma,\mathsf{I},\mathsf{J})$. Based on *COND-SEMANTICS* module we can introduce the desired goal to the ITP with this command:

```
load drex.maude .
load itp-tool.maude .
loop init-itp .
(goal apply-cond : COND-SEMANTICS
   |- A{E:Expr ; S:String ; I:Nat ; J:Nat}
   (((apply(conditional(E,E), S, I, J)) = (apply(E, S, I, J)))) .)
```

Figure 4.1: The Idempotency Goal.

The obtained result of evaluating Example 1 using ITP is illustrated in Figure 4.2.

```
==================================
label-sel: apply-cond@0
==================================
A{E:Expr ; I:Nat ; J:Nat ; S:String}
apply(conditional(E:Expr,E:Expr),S:String,I:Nat,J:Nat)
= apply(E:Expr,S:String,I:Nat,J:Nat)
++++++++++++++++++++++++++++++++++
```

Figure 4.2: The Idempotency Goal Evaluation in ITP.

After that, we can apply structural induction on E to prove the idempotency property of the desired goal as illustrated in Figure 4.3

```
Maude> (ind on E:Expr .)
rewrites: 174 in 32ms cpu (96ms real) (5437 rewrites/second)
Eliminated current goal.

q.e.d

++++++++++++++++++++++++++++++++
```

Figure 4.3: The Idempotency Goal Proved.

The following particular example illustrate the idempotency property of the conditional combinator:

```
Maude> red apply(conditional(CaesarDecryption,CaesarDecryption),
                 "GDEEGF",2,4) .
reduce in FUNCTION-LIB : apply(conditional(CaesarDecryption,
                           CaesarDecryption),"GDEEGF", 2, 4) .
rewrites: 101 in 4ms cpu (0ms real) (25250 rewrites/second)
result String: "BB"
```

Figure 4.4: Evaluation of the Conditional Combinator

```
Maude> red apply(CaesarDecryption,"GDEEGF",2,4) .
reduce in FUNCTION-LIB : apply(CaesarDecryption, "GDEEGF", 2, 4) .
rewrites: 100 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "BB"
```

Figure 4.5: Evaluation of RST Function

```
Maude> red apply(conditional(CaesarDecryption,CaesarDecryption),
           "GDEEGF",2,4) == apply(CaesarDecryption,"GDEEGF",2,4) .
reduce in FUNCTION-LIB : apply(conditional(CaesarDecryption,
           CaesarDecryption),"GDEEGF", 2, 4) == apply(CaesarDecryption,
                                                "GDEEGF", 2, 4) .
rewrites: 193 in 0ms cpu (0ms real) (~ rewrites/second)
result Bool: true
```

Figure 4.6: The Desired Goal

- **Proving Associativity Property of the Conditional Combinator**

Associative operations are abundant in mathematics including addition and multiplication operations. The following functional notation represent the associativity: $\mathsf{f}(\mathsf{f}(\mathsf{x},\mathsf{y}),\mathsf{z}) = \mathsf{f}(\mathsf{x},\mathsf{f}(\mathsf{y},\mathsf{z}))$.

Accordingly, we want to prove that the conditional combinator in the module

```
fmod COND-SEMANTICS is
        inc SEM-INTERFACE .
        vars I J       : Nat . vars E1 E2     : Expr .
        vars S SE1 SE2 : String .
        eq domain(conditional(E1,E2)) = domain(E1) + domain(E2) .
        ceq apply(conditional(E1,E2),S,I,J) = SE1
        if SE1 := apply(E1,S,I,J) .
        ceq apply(conditional(E1,E2),S,I,J) = SE2
        if not (apply(E1,S,I,J) :: String)
        /\ SE2 := apply(E2,S,I,J) .
endfm
```

Figure 4.7: The Conditional Combinator Semantics

satisfies the associativity property by induction using ITP tool.

$(\forall \mathsf{E1},\mathsf{E2},\mathsf{E3})\ \mathsf{cond}(\mathsf{cond}(\mathsf{E1},\mathsf{E2}),\mathsf{E3},\sigma[\mathsf{i},\mathsf{j}]) \ \equiv\ \mathsf{cond}(\mathsf{E1},\mathsf{cond}(\mathsf{E2},\mathsf{E3}),\sigma[\mathsf{i},\mathsf{j}])$

For a Maude module representing the conditional combinator in Figure 4.7, we would like to automatically prove the associativity of the conditional combinator. First, we load into Maude the desired module, then load $\mathsf{itp}-\mathsf{tool}$, after that type $\mathsf{loop\ init}-\mathsf{itp}$, then enter the desired goal. The following Figure illustrates these steps.

```
load drex.maude .
load itp-tool.maude .
loop init-itp .
(goal cond-assoc : COND-SEMANTICS
|- A{E1:Expr ; E2:Expr ; E3:Expr ; I:Nat ; J:Nat ; S:String}
((apply(conditional(conditional(E1,E2),E3),S,I,J)) =
(apply(conditional(E1,conditional(E2,E3)),S,I,J))) .)
```

Figure 4.8: The Associativity Goal.

The obtained result of evaluating Example 2 is illustrated in Figure 4.9

```
==================================
label-sel: cond-assoc@0
==================================
A{E1:Expr ; E2:Expr ; E3:Expr ; I:Nat ; J:Nat ; S:String}
apply(conditional(conditional(E1:Expr,E2:Expr),E3:Expr),
S:String,I:Nat,J:Nat) = apply(conditional(E1:Expr,
conditional(E2:Expr,E3:Expr)),S:String,I:Nat,J:Nat)

++++++++++++++++++++++++++++++++
```

Figure 4.9: The Associativity Goal Evaluation in ITP.

We can then try to prove the goal by *induction* on $\mathsf{E3 : Expr}$ by giving the command ($\mathsf{ind\ on\ E3 : Expr}$). The ITP tool tries to simplify the goal by applying the equation in the module, until reaching an identity,thus proving the associativity property of the combine combinator.

```
Maude> (ind on E3:Expr .)
rewrites: 208 in 20ms cpu (182ms real) (10400 rewrites/second)
Eliminated current goal.

q.e.d

++++++++++++++++++++++++++++++++
```

Figure 4.10: The Associativity Goal Proved.

The following particular example illustrates the associativity property of the

conditional combinator:

```
Maude> red apply(conditional(CaesarEncryption,
conditional(CaesarDecryption,UpperCase)),"abcd",0,2) .
reduce in FUNCTION-LIB : apply(conditional(CaesarEncryption,
conditional(CaesarDecryption, UpperCase)), "abcd", 0, 2) .
rewrites: 317 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "AB"
```

Figure 4.11: (A) Evaluation of the Conditional Combinator

```
Maude> red apply(conditional(conditional(CaesarEncryption,
CaesarDecryption),UpperCase),"abcd",0,2) .
reduce in FUNCTION-LIB : apply(conditional(conditional
(CaesarEncryption,CaesarDecryption), UpperCase), "abcd", 0, 2) .
rewrites: 415 in 0ms cpu (0ms real) (~ rewrites/second)
result String: "AB"
```

Figure 4.12: (B) Evaluation of the Conditional Combinator

```
Maude> red apply(conditional(CaesarEncryption,conditional
                (CaesarDecryption,UpperCase)),"abcd",0,2) ==
           apply(conditional(conditional(CaesarEncryption,
                   CaesarDecryption),UpperCase),"abcd",0,2) .
reduce in FUNCTION-LIB : apply(conditional(CaesarEncryption,
    conditional(CaesarDecryption, UpperCase)), "abcd", 0, 2) ==
    apply(conditional(conditional(CaesarEncryption, CaesarDecryption),
    UpperCase), "abcd", 0, 2)
.
rewrites: 706 in 4ms cpu (1ms real) (176500 rewrites/second)
result Bool: true
```

Figure 4.13: Particular Example of Associativity Property

CHAPTER 5

NDREX: SPECIFICATION OF NON-DETERMINISTIC REGULAR STRING TRANSFORMATIONS

In this chapter, we focus on the second research question, which discusses *non-deterministic* regular string transformation where RQ2 is:

- **[RQ2]** How do we define an extension of this formal specification to provide a formal model for *nondeterministic* regular string transformations in an elegant high-level manner that enables us to simulate and analyze their properties?

NDReX further addresses question by developing a non-deterministic specification based on rewriting logic.

Nondeterminism is a widely useful concept that has a significant impact on the theory of computation and has various applications including automata theory. This idea motivated us to devise a set of formal and executable semantics to capture a *non-deterministic regular string transformations* using the Maude framework. This formal definition useful in various domains that utilize *non-deterministic regular string transformations*.

5.1 Introduction to Non-deterministic Semantics using Maude

Non-deterministic notion is a widely used term in automata theory and is used in a number of real applications. Non-deterministic automata are allowed to have two or more transitions containing the same symbol out of one state as shown in Figure 5.1 which illustrates non-deterministic behavior using finite automata. Therefore, we cannot predict the output because there are multiple possible outcomes for each input.

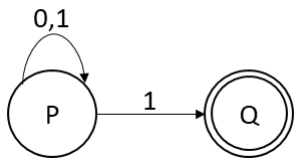

Figure 5.1: Non-deterministic transition from state p on input 1.

Regular expression is an algebraic way to define patterns, it is useful for validating, filtering, classifying input while it has an additional role in capturing

nondeterministic computation by nature. For example, if we have a regular expression $\mathsf{R} := \mathsf{a}|\mathsf{ab}^*$, there are two ways to generate a string "a". Also, regular expression provides a way to capture nondeterministic automata. However, both of them have a limitation in terms of formal analysis.

Rewriting logic can naturally deal with non-deterministic computation and its behavior has been specified in the Maude in terms of rewriting rules corresponding to the model specification (*system module*). In fact, rewriting logic specifications are executable which allows for the application of a wide range of formal analysis methods and tools [21].

Due to Maude's ability to deal with non-deterministic computation, we are able to develop a formal specification of non-deterministic regular string transformations whereas the previous work (DReX [1]) failed to capture non-deterministic regular string transformation.

In DReX, we have analyzed and investigated every form of combinator behavior in order to ascertain whether it displays non-deterministic behavior. We have found that the *split combinator* has been used to capture regular string transformation by making an assumption over a uniquely split the input string (σ), otherwise, the split combinator will fail, thus producing an error ($\perp$) as a result of evaluating regular string transformations. However, in our research, we are able to modify *the split combinator* behavior and eliminate the assumption made by Alur ***et al*** [1]. Therefore, we are able to handle and capture regular string transformation using *the split combinator* with the input string (σ) that has multiple

splits. In doing this, we utilize the rule-based technique, which can be used for expressing and capturing the *non-deterministic behavior of the split combinator*.

5.2 Formal Specification of Non-deterministic Regular String Transformation in Maude

In this section, we show how the non-deterministic behavior of regular string transformation (given by *the split combinator*) can be represented by their corresponding Maude specification.

5.2.1 Split Position Algorithm

The split position algorithm (Algorithm 1, described in Section 3.3.5) has been used to retrieve all possible indices representing valid splits. These positions are equivalent to a non-deterministic transition in finite automata. Therefore, these positions are used by the split combinator to present multiple possible outputs for the same input.

5.2.2 Encoding Split Combinator in Maude using Rewrite Rule

In order to capture deterministic regular string transformations, we used the algebraic semantics of the split combinator, as presented in Chapter 3, and implemented based on equational theory. Modifying this specification to capture a

nondeterministic regular string transformations can be easily done using rewriting logic since the equational logic is a sub-logic of rewriting logic [14].

Consequently, we implemented only one conditional rule in Maude to capture the nondeterministic behavior of the split combinator as illustrated in the following Maude syntax. Furthermore, the syntax and semantics of other combinators do not change.

```
*** Semantics of split
crl [splitRule] : apply(split(E1,E2),S,I,J)
=> S1 + S2
if chkIndex(S,I,J)
/\ S' := substr(S,I, sd(J, I))
/\ L1, K, L2 := splits(S',E1,E2)
/\ S1 := apply(E1, S', 0, K + 1)
/\ S2 := apply(E2, S', K + 1, length(S')) .
```

In this conditional rule, the *matching equations* employed to match the obtained result of evaluating $substr(S, I, sd(J, I))$, $splits(S', E1, E2)$, $apply(E1, S', 0, K + 1)$ and $apply(E2, S', K + 1, length(S'))$ to the string, list of valid indices, string and string, respectively. Moreover, the *splits* equation invoke to retrieve all valid indices represented in list form. The Maude interpreter will pick one element from this list and assign to k.

Deterministic vs. Nondeterministic

To clarify the difference between the deterministic and nondeterminis-

tic behavior of the split combinator, we evaluate the following example; apply(split(CaesarEncryption, LowerCase), "ABCD", 0, 4) using both algebraic semantics and rewriting logic to observe the difference in outcomes.

```
Maude> red apply(split(CaesarEncryption,LowerCase),"ABCD
   ",0,4) .
reduce in FUNCTION-LIB : apply(split(CaesarEncryption,
   LowerCase), "ABCD", 0,
4) .
rewrites: 617 in 0ms cpu (0ms real) (~ rewrites/second)
result [String]: apply(split("CaesarEncryption",(ev("A")
    + ev("B") + ev("C") +
ev("D")) *, "ConvertToLower",(ev("A") + ev("B") + ev("C"
   ) + ev("D")) *),
"ABCD", 0, 4)
```

```
Maude> rew apply(split(CaesarEncryption,LowerCase),"ABCD
   ",0,4) .
rewrite in FUNCTION-LIB : apply(split(CaesarEncryption,
   LowerCase), "ABCD", 0,
4) .
rewrites: 804 in 0ms cpu (1ms real) (~ rewrites/second)
result String: "Dbcd"
```

Since the input string ($\sigma = "ABCD"$) has multiple split positions (multiple

solutions), the algebraic semantics failed to evaluate the expression. However, the rewriting semantics were able to evaluate the expression and produced *Dbcd* as the obtained result.

5.3 Formal Analysis of NDReX with Maude

Once the system specifications were developed in Maude, we obtained a rewriting logic specification of the system. As these specifications are executable, they can be used to analyze and simulate the system and provide a detailed step-by-step formal description of a program's execution [14].

In this work, we focus on two kinds of analyses: *simulation,* to execute the non-deterministic regular string transformation specifications; and *reachability analysis,* to prove system invariants.

5.3.1 Simulation

The **frewrite** and **rewrite** commands execute the Maude Specifications, which implement two different execution strategies: a depth-first position-fair strategy, and a top-down rule-fair strategy, respectively [21]. Thus, we can execute the non-deterministic regular string transformation of the split combinator by simply typing: rew apply(split(CaesarEncryption, LowerCase), "ABCD", 0, 4).

The output of `rew` command results in applying a split over σ that has a multiple split, and produces a string as the obtained output rather than the error ($\perp$).

```
Maude> rew apply(split(CaesarEncryption,LowerCase),"ABCD
   ",0,4) .
rewrite in FUNCTION-LIB : apply(split(CaesarEncryption,
   LowerCase), "ABCD", 0,
4) .
rewrites: 804 in 0ms cpu (1ms real) (~ rewrites/second)
result String: "Dbcd"
```

Also, the **frewrite** and **rewrite** commands allow users to specify the upper bounds for the number of rule applications. This can be very useful performing step-by-step executions, or to simulate non-terminating systems.

5.3.2 Reachability Analysis

Executing the system using the **frewrite** and **rewrite** commands means exploring just one of the possible behaviors of the system. However, the **search** command in Maude allows the exploration of the reachable state space in a variety of ways by following a breadth-first strategy up to a specified bound. The command requires specifying an input, which in this case comprises the properties that the reachable states have to satisfy, and it then returns those states that satisfy them in turn [34].

Also, the **search** command has an additional use in its ability to prove safety properties, which states that something bad should never happen where the arrow

$=> *$ of **search** command implies searching for proofs fulfilling the search pattern consisting of none, one or more rewriting steps. For instance, we can check whether the input string (σ) satisfies the domain of the split combinator and produces a string as an obtained result or not. The Maude **search** command $=> *$ can be used to show the following property:

```
search apply(split(CaesarEncryption,LowerCase),"abcd"
   ,0,4) =>* S:String .
search in FUNCTION-LIB : apply(split(CaesarEncryption,
   LowerCase), "abcd", 0,
4) =>* S .
No solution.
```

Moreover, the **search** command allows us to automatically explore all of the possible solutions using the breadth-first search technique. The following example shows all possible solutions of executing $\mathsf{apply}(\mathsf{split}(\mathsf{CaesarEncryption}, \mathsf{LowerCase}), \text{"}\mathsf{ABCD}\text{"}, 0, 4)$ using Maude search command:

```
Maude> search apply(split(CaesarEncryption,LowerCase),"
   ABCD",0,4) =>* S:String .
search in FUNCTION-LIB : apply(split(CaesarEncryption,
   LowerCase), "ABCD", 0,
4) =>* S .
Solution 1 (state 1)
```

```
states: 2  rewrites: 804 in 0ms cpu (1ms real) (~
   rewrites/second)
S --> "Dbcd"


Solution 2 (state 2)
states: 3  rewrites: 989 in 0ms cpu (1ms real) (~
   rewrites/second)
S --> "DEcd"


Solution 3 (state 3)
states: 4  rewrites: 1172 in 0ms cpu (2ms real) (~
   rewrites/second)
S --> "DEFd"


No more solutions.
states: 4  rewrites: 1374 in 0ms cpu (2ms real) (~
   rewrites/second)
```

The above example represent a non-deterministic regular string transformations. To handle this situation and explore all possible solutions we use breadth-first search technique by executing the **search** command where we found three possible solutions. The First solution obtained by applying $CaesarEncryption$ over A and $LowerCase$ over BCD and the result is "$Dbcd$. We obtained $DEcd$ as

a second solution by applying $CaesarEncryption$ over AB and $LowerCase$ over CD. Finally, the **search** command apply $CaesarEncryption$ and $LowerCase$ over ABC and D, respectively and the obtained result is $DEFd$.

CHAPTER 6

RELATED WORK

In this chapter, we briefly review related work and highlight some essential information in the different areas to which this thesis contributes.

6.1 Regular String Transformations Models

There are three equivalent models that have been used to capture regular string transformation. These models are the two-way deterministic automata with output, the monadic second order (MSO) transduction and the streaming transducers with register [35]. Therefore, any string to string transformation that can be realized by one of these models is called a regular string transformation model.

Alur ***et al*** (2014) [4] proposed a set of combinators to capture regular string to string transformation for document processing. These combinators are (i) base combinator $L/d(\sigma)$, (ii) conditional choice combinator $f \triangleright g(\sigma)$, (iii) split sum combinator $f \oplus g(\sigma)$, (iv) left-split sum combinator $f \overleftarrow{\oplus} g(\sigma)$, (v) iterate sum combinator $\Sigma f(\sigma)$, (vi) left-iterate sum combinator $\overleftarrow{\Sigma} f(\sigma)$ and (vii) sum combinator

$f(\sigma) + g(\sigma)$. The proposed combinators have the similar meaning of regular expression on regular language such that conditional combinator is similar to union, split sum combinator is analogous to concatenation and iterate sum combinator is equivalent to Kleene-*. The main contribution of this research was proof that all regular functions can be inductively generated from these combinators.

Alur and Černỳ (2011) [36] proposed a new abstract and analyzable model which called streaming data string transducer (SST) used in a single-pass list processing program. The proposed model is a deterministic transducer with string variables to capture string transformation where input data strings map to output data strings. Furthermore, SST model has been used to check functional equivalence problem, checking correctness with respect to pre/post conditions, and assertion checking problem. Moreover, SST model leads to algorithms for checking functional equivalence of two programs that are written in different programming paradigm (functional and imperative style) for processing lists of data item.

Engelfriet and Hoogeboom (2001) [37] studied deterministic regular string transduction and extend a classic result of monadic string second order (MSO) definable string transduction which is a type of logical framework that uses quantification over sets of positions in the string to define string properties. Also, they proposed a hybrid model that allows two-way machines to jump to new positions on the tape as specified by an MSO formula and they demonstrated the equivalence between this model and the basic two-way machine model.

6.2 Domain Specific Language for String transformations

There are two types of domain specific languages that are used to capture string transformation. These are transducer-based language and string specific utilities such as Perl, sed and AWK [1].

Perl, AWK and sed are domain-specific programming languages used for string transformation and widely employed to reformat and query text file. However, these languages do not support algorithmic analysis [1, 38].

Hooimeijer ***el al.*** (2011) [39] proposed a domain-specific programming language for modeling string transformation called BEK. The symbolic finite transducer has been used in BEK [40]. Moreover, BEK is a language and a complier used for writing and analyzing string manipulation. The web applications often use special string transformation sanitizers on untrusted user data where the sanitizer is a function that removes or escapes potentially dangerous strings. Therefore, BEK has been used for writing sanitizer to perform security specific analysis and capture real web sanitizers such as Google AutoEscape sanitizer and IE XSS Filter. The *symbolic finite automata* representation has been used by BEK in order to perform required analysis such as to determine if two BEK programs are equivalent and if a BEK program can output a specific string.

D'Antoni ***el al.*** (2014) [41] presented a domain-specific programming language based on *Symbolic Tree Transducer* with Regular lookahead (STTR) named FAST (Functional Abstraction of symbolic Transducers) for string and tree manipula-

tion. FAST could be used for modeling and analyzing programs that manipulate strings and tree over infinite domains. FAST has been used to model numerous real world scenarios and analysis problems such as analysis functional programs over list and trees.

6.3 Executable Formal Semantics

There are numerous studies that cover formal method to present formal semantic handling by programming languages. However, without formal semantics it is difficult to prove or state anything about the programing languages such as whether a certain program meets its specification or the interpreter or compiler is correct, etc. Also, formal semantics provide the ability to use formal analysis tools in order to minimize mistakes in semantics. In this section, we covered various works that discuss and present executable formal semantic for different programing languages using formal method and rewriting logic engines.

Bogdanas and Roşu (2015) [42] proposed a complete formal semantics of Java using $\mathbb{k}$ semantic framework. This semantics cover all official definition features of Java 1.4. The proposed semantics split into (a) static semantic (b) dynamic semantic. The static semantics takes as input the abstract syntax tree representation of Java program and produced a valid Java program that contains different Java features then it passed to dynamic semantic for execution (see Section 4 in [42]). Furthermore, Test Driven Development methodology has been used by the authors to verify and validate their semantic and to emphasize the completeness

and useful of this semantics, the built-in model-checker of K has been used to check multi-threading Java programs.

Guth (2013) [43] worked on $\mathbb{k}$ rewriting semantic framework to develop a formal operational semantic of Python as a thesis work. The proposed semantics is executable but incomplete. However, Python doesn't have a complete formal semantics. This semantics has been thoroughly tested against a large library of unit test.

Ellison and Roşu (2012) [44] presented an executable formal definition of C. This formal semantic can be used to find program bugs and captures all feature required by the C99. They used a rewriting framework called $\mathbb{k}$ to develop their semantics and the proposed semantic written using the K-Maude tool. Moreover, different types of test have been used to verify the correctness of this semantic such as correctness analysis and exploratory testing, etc.(see Section 5 in [44]).

Rivera ***et al*** (2009) [31] extended a previous work to propose an efficient formal specification of behavioral semantics for domain specific models (DSMs) using Maude. DSM is an approach to design and develop systems by using domain specific languages (DSLs) to describe different features of a system in terms of models. However, there are numerous approaches to express the behavioral semantics of meta-models. These approaches have limitations in order to capture complex behavior at high level of abstraction. Another limitation proposed approach does not support the proper tool for formal analysis. Also, DSM community doesn't give much attention to the behavioral semantics of models. In addition, modeling

languages play an essential role in model driven software development. Theses factors motivate the authors to present a formal specification of the behavioral semantics of meta-models. They used Maude as a formal representation for defining models, meta-models and their dynamic behavior. Also, Maude can be used to specify abstract syntax and semantics.

Meredith ***et al*** (2007) [11] proposed a formal executable semantics of K-Scheme (An algebraic denotational specification). This executable semantic was the most complete formal definition of a language in the scheme. Furthermore, Maude rewriting logic engine has been used by the authors to deploy this semantic. The authors focused on Scheme as a programing language because Scheme employing pattern describes syntax transformation.

6.4 Equational Specification of Regular Languages

Roşu and Viswantathan (2003) [33] proposed a rewriting based algorithm for testing membership of a word in regular language defined by an extended regular expression. The main concern of this algorithm is to tackle the membership problem where regular expression is useful to specify patterns in strings. Membership problem is a problem to decide whether a word w is in the regular language generated by regular expression R. There are many applications have been developed to solve membership problem. However, regular expression has been used in [33] due

to their power in specifying patterns. The extended regular expression presented in [33] added complementation $\neg R$ to usual regular expression operations. The proposed algorithm was implemented and evaluated using Maude framework.

Antimirov and Mosses (1995) [45] proposed an extended algebra of regular language that capture intersection operation in addition to the usual operations on a given alphabet. Horn-equational axiomatization has been used to represent a new collection of equations that capture the extended algebra of regular languages. OBJ rewrite system has been used by the authors to deploy some term-rewriting techniques in order to prove/disprove equations between extended regular expressions.

To best of our knowledge this is the most appropriate formal specification that capture both *deterministic* and *non-deterministic* regular string transformations where most of other works in the literature focused on *deterministic*. Also, the related works captured regular string transformations through a complex models such as transducer which have a limitation in terms of formal analysis. Moreover, to prove some properties such as idempotency or associativity required further manual analysis using a pencil and paper, it takes time and expensive to build and maintain, also it is difficult to proof theses properties using transducer models. However, in our work we used an automated tool to proof these properties that emphasize the correctness of our work and save our time through this thesis.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 Conclusion

In this thesis, we present formal executable semantics for regular string transformation, based on algebraic specification (deterministic behavior) and rewriting logic (non-deterministic behavior), and carry out it's implementation in Maude framework. This semantics is executable and has been used to implement different examples to simulate regular string transformation behavior. Further, we performed formal analysis and verification, such as simulation and reachability analysis.

7.2 Limitation

The current developed specification handles only one character at a time based on Alur ***et. al*** in [1]. This allowed our specification a natural effectiveness such that it cannot be intuitively decomposed into anything simpler and all the functions can be easily expressed using this single character technique. However, this resulted in more processing time for input string (σ) of longer lengths which affects the efficiency for such cases.

7.3 Future Work

In this section, we would discuss recommendations to further improve this work and implementation of the proposed formal specification of regular string transformation for practical applications.

7.3.1 Build a Tool for an IDE software

One possible way to use our specification for practical application is to build a tool or plug-in for IDE softwares such as *Eclipse*. This tool can serve as a platform to perform both formal analysis and simulation based on regular string transformation. Moreover, it can also be used by any researcher or developer without prior knowledge of Maude syntax. Recently, such a tool has already been developed by Bogdanas and Roşu in [42] for K-Java.

7.3.2 Utilization in Real Application

As mentioned in the introductory chapter, the security employs regular string transformation and has been using it in several application such as encryption. The proposed formal specification has mathematically proved itself as a reliable and robust method for regular string transformation. Therefore, our specification can be used to formally present a solution for security relating problems and perform formal analysis to enhance and improve the security of applications.

7.3.3 Improving Efficiency

The presented specification deals with single character to perform regular string transformation as mentioned in the limitation section . Hence, this specification can be improved to deal with a chunk of strings for longer input string (σ) which will further enhance the performance and efficiency of the specification.

REFERENCES

[1] R. Alur, L. D'Antoni, and M. Raghothaman, "Drex: A declarative language for efficiently evaluating regular string transformations," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 125–137.

[2] N. Bjørner, N. Tillmann, and A. Voronkov, "Path feasibility analysis for string-manipulating programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2009, pp. 307–321.

[3] Z. Wang, G. Xu, H. Li, and M. Zhang, "A probabilistic approach to string transformation," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 5, pp. 1063–1075, 2014.

[4] R. Alur, A. Freilich, and M. Raghothaman, "Regular combinators for string transformations," in *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. ACM, 2014, p. 9.

[5] J. Weinberger, P. Saxena, D. Akhawe, M. Finifter, R. Shin, and D. Song, "A systematic analysis of xss sanitization in web application frameworks," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 150–171.

[6] D. Hofheinz, J. Malone-Lee, and M. Stam, "Obfuscation for cryptographic purposes," in *Theory of Cryptography Conference*. Springer, 2007, pp. 214–232.

[7] R. Alur, E. Filiot, and A. Trivedi, "Regular transformations of infinite strings," in *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*. IEEE Computer Society, 2012, pp. 65–74.

[8] M. Raghothaman, "regular programming over data streams," 2015.

[9] M. Mohri, "Finite-state transducers in language and speech processing," *Computational linguistics*, vol. 23, no. 2, pp. 269–311, 1997.

[10] J. Meseguer, "Twenty years of rewriting logic," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 7, pp. 721–781, 2012.

[11] P. Meredith and M. H. G. Rosu, "An executable rewriting logic semantics of k-scheme," in *Workshop on Scheme and Functional Programming*, vol. 1, no. 2007/9, 2007, p. 10.

[12] G. Rou, "From rewriting logic, to programming language semantics, to program verification," in *Logic, Rewriting, and Concurrency*. Springer, 2015, pp. 598–616.

[13] M. A. Al-Turki, "Rewriting-based formal modeling, analysis and implementation of real-time distributed services," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 2011.

[14] J. Meseguer and G. Roşu, "Rewriting logic semantics: From language specifications to formal analysis tools," in *International Joint Conference on Automated Reasoning*. Springer, 2004, pp. 1–44.

[15] A. Farzan, F. Chen, J. Meseguer, and G. Roşu, "Formal analysis of java programs in javafan," in *International Conference on Computer Aided Verification*. Springer, 2004, pp. 501–505.

[16] A. Verdejo and N. Martí-Oliet, "Executable structural operational semantics in maude," *The Journal of Logic and Algebraic Programming*, vol. 67, no. 1-2, pp. 226–293, 2006.

[17] J. A. V. López and N. M. Oliet, "Executable structural operational semantics in maude," 2003.

[18] G. Kahn, "Natural semantics," *STACS 87*, pp. 22–39, 1987.

[19] G. D. Plotkin, "A structural approach to operational semantics," 1981.

[20] J. Meseguer, "Conditional rewriting logic as a unified model of concurrency," *Theoretical computer science*, vol. 96, no. 1, pp. 73–155, 1992.

[21] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All about maude-a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. F. Quesada, "Maude: specification and programming in rewriting logic," *Theoretical Computer Science*, vol. 285, no. 2, pp. 187–243, 2002.

[23] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martı-Oliet, J. Meseguer, and J. Quesada, "A maude tutorial," *Computer Science Laboratory, SRI International*, 2000.

[24] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, "Principles of maude," *Electronic Notes in Theoretical Computer Science*, vol. 4, pp. 65–89, 1996.

[25] M. Clavel, F. Durán, J. Hendrix, S. Lucas, J. Meseguer, and P. Ölveczky, "The maude formal tool environment," in *International Conference on Algebra and Coalgebra in Computer Science*. Springer, 2007, pp. 173–178.

[26] M. A. AlTurki and J. Meseguer, "Executable rewriting logic semantics of orc and formal analysis of orc programs," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 4, pp. 505–533, 2015.

[27] M. Clavel and M. Palomino, "The itp tool," in *Logic, Language and Information. Proceedings of the First Workshop on Logic and Language*, 2001, pp. 55–62.

[28] M. Clavel, M. Palomino, and A. Riesco, "Introducing the itp tool: a tutorial." *J. UCS*, vol. 12, no. 11, pp. 1618–1650, 2006.

[29] J. Hendrix, J. Meseguer, and R. Sasse, "Maude itp 2.0 tutorial," Technical report, University of Illinois at Urbana-Champaign, Tech. Rep., 2008.

[30] P. C. Ölveczky and M. Caccamo, "Formal simulation and analysis of the cash scheduling algorithm in real-time maude," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2006, pp. 357–372.

[31] J. E. Rivera, F. Durán, and A. Vallecillo, "Formal specification and analysis of domain specific models using maude," *Simulation*, 2009.

[32] N. Martı-Oliet and J. Meseguer, "Rewriting logic: roadmap and bibliography," *Theoretical Computer Science*, vol. 285, no. 2, pp. 121–154, 2002.

[33] G. Roşu and M. Viswanathan, "Testing extended regular language membership incrementally by rewriting," in *International Conference on Rewriting Techniques and Applications*. Springer, 2003, pp. 499–514.

[34] J. E. Rivera, E. Guerra, J. de Lara, and A. Vallecillo, "Analyzing rule-based behavioral semantics of visual modeling languages with maude," in *International Conference on Software Language Engineering*. Springer, 2008, pp. 54–73.

[35] M. Bojańczyk, "Transducers with origin information," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2014, pp. 26–37.

[36] R. Alur and P. Černỳ, "Streaming transducers for algorithmic verification of single-pass list-processing programs," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 599–610.

[37] J. Engelfriet and H. J. Hoogeboom, "Mso definable string transductions and two-way finite-state transducers," *ACM Transactions on Computational Logic (TOCL)*, vol. 2, no. 2, pp. 216–254, 2001.

[38] R. Alur, D. Fisman, and M. Raghothaman, "regular programming for quantitative properties of data streams," in *European Symposium on Programming Languages and Systems*. Springer, 2016, pp. 15–40.

[39] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes, "Fast and precise sanitizer analysis with bek," in *Proceedings of the 20th USENIX conference on Security*. USENIX Association, 2011, pp. 1–1.

[40] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner, "Symbolic finite state transducers: Algorithms and applications," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 137–150, 2012.

[41] L. D'Antoni, M. Veanes, B. Livshits, and D. Molnar, "Fast: A transducer-based language for tree manipulation," in *ACM SIGPLAN Notices*, vol. 49, no. 6. ACM, 2014, pp. 384–394.

[42] D. Bogdanas and G. Roşu, "K-java: a complete semantics of java," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 445–456.

[43] D. Guth, "A formal semantics of python 3.3," 2013.

[44] C. Ellison and G. Rosu, "An executable formal semantics of c with applications," in *ACM SIGPLAN Notices*, vol. 47, no. 1. ACM, 2012, pp. 533–544.

[45] V. M. Antimirov and P. D. Mosses, "Rewriting extended regular expressions," *Theoretical Computer Science*, vol. 143, no. 1, pp. 51–72, 1995.

APPENDIX A
Algebraic Specification in Maude

```
***(
This file is part of Algebraic-DReX, an equational logic
semantics of DReX, a language for regular string transfo-
rmations, with formal analysis through equational simpli-
fication and inductive theorem proving.

Copyright (C) 2015-2017 Shadi A. Alhaj (g201408500@kfupm.edu.sa)
and Musab A. Alturki, musab.alturki@gmail.com

This program is free software: you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the Free Software Foundation, either version 3 of
the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public
License along with this program.
If not, see <http://www.gnu.org/licenses/>.
)

****                              Main file and modules
****                              Authors: Shadi A. Alhaj
****                                       Musab A. Alturki
```

```
*** This is an adaptation of Sen and Rosu's algebraic specif-
*** ication of ERE's with a simplified axiomatization of regular
*** expressions using subsorting

fmod RE is
        *** NeeRe: A regular expression that is
        *** neither empty nor epsilon
        sorts Event NeeRe Re .
        subsort Event < NeeRe < Re .

        *** the singlton language containing only epsilon
        op epsilon : -> Re .

        *** the empty language
        op empty : -> Re .

        *** Alternation (the union of two regular languages)
        op _+_ : Re    Re -> Re    [assoc comm id: empty prec 60] .
        op _+_ : NeeRe Re -> NeeRe [ditto] .

        *** Concatenation
        op __  : Re    Re    -> Re  [assoc id: epsilon prec 50] .
        op __  : NeeRe NeeRe -> NeeRe [ditto] .

        *** Kleene star
        op (_*) : Re    -> Re    [prec 40] .
        op (_*) : NeeRe -> NeeRe [ditto] .
endfm


*** Equational simplication of RE, a heavy adaptation of
*** Sen and Rosu's specifications

fmod SIMPLIFY_RE is
        inc RE .

        var R  : Re .
        var NR : NeeRe .

        *** idempotency of +
        eq NR      + NR      = NR .
        eq epsilon + epsilon = epsilon .

        *** empty: absorbing element for concatenation
        eq empty NR    = empty .
        eq NR    empty = empty .
        eq empty empty = empty .
```

```
        *** kleene star properties
        eq R * *     = R * .
        eq epsilon * = epsilon .
        eq empty *   = epsilon .
endfm

*** Membership in RE's, a heavy adaptation of
*** Sen and Rosu's specifications

fmod RE_MEMBERSHIP is
        pr SIMPLIFY_RE .

        *** Membership operation
        op _in_ : Re Re -> Bool [prec 80] .

        vars R R1 R2                : Re .
        vars NR NR1 NR2 NR1' NR2' : NeeRe .
        vars A B                    : Event .

        *** general properties
        eq empty in R = true .
        eq R     in R = true .

        *** membership over +
        eq R in NR1 + NR2     = (R in NR1) or (R in NR2) .
        eq R in NR1 + epsilon = (R in NR1) or (R in epsilon) .

        *** membership over concatenation
        eq epsilon in (NR1 NR2)
        = (epsilon in NR1) and (epsilon in NR2) .
        eq A in (NR1 NR2)
        = (A in NR1) and (epsilon in NR2) or
          (A in NR2) and (epsilon in NR1) .
        ceq NR1 NR2 in NR1' NR2'
        = true if NR1 in NR1' /\ NR2 in NR2' .

        *** membership over *
        eq epsilon in R *   = true .
        eq A in NR *        = A in NR .
        ceq NR in NR *       = true if NR in NR .
        ceq NR1 NR2 in NR *  = true if NR1 in NR
                                    /\ NR2 in NR * .
        *** If the RE's fail matching the cases above,
        *** then membership fails
        eq R1 in R2 = false [owise] .
endfm
```

```
*** A date type for lists of positions (Nats)

fmod POS-LIST is
        pr NAT .

        *** Lists of Nats: comma-separated lists of naturals
        sort List .
        subsort Nat < List .

        *** the empty list
        op mt : -> List .
        op _,_ : List List -> List [assoc id: mt] .

        var N : Nat . var L : List .

        *** List size
        op size : List -> Nat .
        eq size(mt) = 0 .
        eq size(N,L) = 1 + size(L) .
endfm

*** An abstract representation of a regular
*** transformation function

fmod FUNCTION is
        pr STRING .
        pr RE_MEMBERSHIP .

        sort Function .

        *** A function as a pair of a name and a domain
        op (_,_) : String Re -> Function .

        vars S : String .
        var  R   : Re .

        *** The domain of a function
        op domain : Function -> Re .
        eq domain(S,R) = R .

        *** The name of a function
        op fName : Function -> String .
        eq fName(S,R) = S .

        *** An interface for a function's behavior
        op applyFunction : String String -> String .
endfm
```

```
*** Auxiliary operators

fmod FUNCTION_AUX is
        pr FUNCTION .

        *** Converts a string to a regular expression
        op ev : String ~> Re .

        *** This is important typing info
        op ev : Char -> Event .

        var  S : String .
        var  F : Function .
        vars I J : Nat .

        ceq ev(S)  = ev(substr(S,0,1))ev(substr(S,1,length(S)))
                     if length(S) > 1 .
        eq ev("") = epsilon .

        *** check membership of a string in
        *** the domain of a function
        op chkDomain : Function String -> Bool .
        eq chkDomain(F, S) = ev(S) in domain(F) .

        *** check that the indices I and J are
        *** valid w.r.t. the given string
        op chkIndex : String Nat Nat -> Bool .
        eq chkIndex(S,I,J)
        = not( I >= length(S) or J > length(S) or J < I ) .
endfm

*** Syntax of DReX

fmod DREX-SYNTAX is
        inc FUNCTION_AUX .

        sorts Combinator Expr .
        subsort Function Combinator < Expr .

        *** base combinators:
        *** epsilon (parametric) and bottom
        op bot :        -> Combinator .
        op eps : String -> Combinator .

        *** main combinators
        op combine     : Expr Expr -> Combinator .
        op conditional : Expr Expr -> Combinator .
```

```
        op iterate     : Expr      -> Combinator .
        op split       : Expr Expr -> Combinator .
endfm

*** The semantics interface that need to
***be defind for all combinators

fmod SEM-INTERFACE is
        inc DREX-SYNTAX .

        var S : String .
        var E : Expr .

        *** evaluates a DReX expression
        op apply : Expr String Nat Nat ~> String .

        *** The domain of an expression
        op domain : Expr ~> Re .

        *** checks whether a string belongs
        *** to the domain of an expression
        op chkDomain : Expr String -> Bool .
        eq chkDomain(E, S) = ev(S) in domain(E) .
endfm

*** Semantics of the base combinators

fmod BASE-SEMANTICS is
        inc SEM-INTERFACE .

        var  F      : Function .
        vars I J    : Nat .
        vars S S' A : String .

        *** bot is always undefined,
        *** so no equation for its apply
        *** apply
        eq domain(bot) = empty .

        *** epsilon is defined only
        *** if the input string is ""
        eq apply(eps(S),"",0,0) = S .
        eq domain(eps(S)) = ev(S) .

        *** evaluation of functions
        ceq apply(F,S,I,J) = applyFunction(A, S')
        if chkIndex(S,I,J)
```

```
        /\ S' := substr(S,I, sd(J,I))
        /\ chkDomain(F, S')
        /\ A := fName(F) .
endfm

*** Semantics of the combine combinator

fmod COMBINE-SEMANTICS is
        inc SEM-INTERFACE .

        vars I J        : Nat .
        vars E1 E2      : Expr .
        vars S SE1 SE2 : String .

        *** The domain of combine (assumes consistent DReX!)
        ceq domain(combine(E1,E2))
        = domain(E1) if domain(E1) == domain(E2) .

        *** Semantics of combine
        ceq apply(combine(E1,E2),S,I,J) = SE1 + SE2
        if domain(combine(E1,E2)) :: Re
        /\ SE1 := apply(E1,S,I,J)
        /\ SE2 := apply(E2,S,I,J) .
endfm

*** Semantics of the conditional combinator

fmod COND-SEMANTICS is
        inc SEM-INTERFACE .

        vars I J        : Nat .
        vars E1 E2      : Expr .
        vars S SE1 SE2 : String .

        *** the domain of conditional
        eq domain(conditional(E1,E2)) = domain(E1) + domain(E2) .

        *** Semantics of conditional
        ceq apply(conditional(E1,E2),S,I,J) = SE1
        if SE1 := apply(E1,S,I,J) .
        ceq apply(conditional(E1,E2),S,I,J) = SE2
        if not (apply(E1,S,I,J) :: String)
        /\ SE2 := apply(E2,S,I,J) .
endfm

*** Semantics of the iterate combinator
```

```
fmod ITER-SEMANTICS is
        inc SEM-INTERFACE .

        vars F    : Function .
        vars I J  : Nat .
        vars S S' : String .
        var  C    : Char .
        var  E    : Expr .

        *** domain of iterate
        eq domain(iterate(E)) = domain(E)* .

        *** Semantics of iterate
        *** Due to iterate functionality we used
        *** apply(iterate(F),Sigma,0,1)
        *** to evaluate one char each time

        *** Base case when the string is empty
        eq apply(iterate(F),"",I,J) = "" .

        *** Base case when the string is
        *** just one character
        eq apply(iterate(F),C,I,J) = apply(F,C,I,J) .

        *** Inductive case when the length
        *** is greater than one
        ceq apply(iterate(F),S,I,J) = apply'(F,S')
        if length(S) > 1
        /\ chkIndex(S,I,J)
        /\ S' := substr(S,I,sd(J, I))
        /\ chkDomain(iterate(F),S') .

        *** a helper function
        op apply' : Expr String -> String .
        eq apply'(F,"") = "" .
        eq apply'(F, C) = apply(F,C,0,1) .
        eq apply'(F, S) = apply(F,S,0,1) +
                          apply'(F,substr(S,1,length(S))) [owise] .
endfm

*** Semantics of the split combinator

fmod SPLIT-SEMANTICS is
        inc SEM-INTERFACE .
        pr POS-LIST .
        vars S S1 S2 S'  : String .
        vars E1 E2       : Expr .
```

```
        vars BinD1 BinD2 : Bool .
        vars K I J       : Nat .
        vars L1 L2       : List .

        *** Domain of a split expression
        eq domain(split(E1,E2)) = domain(E1) domain(E2) .

        *** Semantics of split
    ceq apply(split(E1,E2),S,I,J) = S1 + S2
            if chkIndex(S,I,J)
            /\ S' := substr(S,I, sd(J, I))
            /\ K  := splits(S',E1,E2)
            /\ S1 := apply(E1, S', 0, K + 1)
            /\ S2 := apply(E2, S', K + 1, length(S')) .

        *** The split position(s) in
        *** a string given two expressions
        op splits : String Expr Expr -> Nat .
        eq splits(S, E1, E2) = splitsPosition(S, 0, E1, E2) .

        *** a helper function for the
        *** split position(s) algorithm
        op splitsPosition : String Nat Expr Expr -> List .
        ceq splitsPosition(S,K,E1,E2) = mt if K > length(S) .
        ceq splitsPosition(S,K,E1,E2)
            = if BinD1 and BinD2 then
                  K, splitsPosition(S, K + 1,E1, E2)
              else
                  splitsPosition(S, K + 1, E1, E2)      fi
        if K <= length(S)
        /\ BinD1 := chkDomain(E1,substr(S,0,K + 1))
        /\ BinD2 := chkDomain(E2,substr(S,K + 1,length(S))) .
endm

*** DReX semantics

fmod DREX-SEMANTICS is
        pr BASE-SEMANTICS .
        pr COMBINE-SEMANTICS .
        pr COND-SEMANTICS .
        pr ITER-SEMANTICS .
        pr SPLIT-SEMANTICS .
endm
```

APPENDIX B
Delete One-Line Comment Specification in Maude

```
*** Delete one-line comments from a program
fmod DEL-FUNCTION is
        inc FUNCTION_AUX .
        var S : String .

        *** To delete "/" from a string
        op del-slashes : -> Function .
        eq del-slashes = ("DeleteSlashes",ev("/")) .
        eq applyFunction("DeleteSlashes",S) = ""   .

        *** To delete "\n" from a string
        op del-nl : -> Function .
        eq del-nl = ("DeleteNewLine",ev("\n")) .
        eq applyFunction("DeleteNewLine",S) = ""   .

        *** To delete anyChar =/= "\n"
        op del : -> Function .
        eq del = ("DeleteAnyChar",
        (ev("a") + ev("b") + ev("c") + ev("d") + ev("e") +
         ev("f") + ev("g") + ev("h") + ev("i") + ev("j") +
         ev("k") + ev("l") + ev("m") + ev("n") + ev("o") +
         ev("p") + ev("q") + ev("r") + ev("s") + ev("t") +
         ev("u") + ev("v") + ev("w") + ev("x") + ev("y") +
         ev("z") + ev("A") + ev("B") + ev("C") + ev("D") +
         ev("E") + ev("F") + ev("G") + ev("H") + ev("I") +
         ev("J") + ev("K") + ev("L") + ev("M") + ev("N") +
         ev("O") + ev("P") + ev("Q") + ev("R") + ev("S") +
         ev("T") + ev("U") + ev("V") + ev("W") + ev("X") +
         ev("Y") + ev("Z") + ev("1") + ev("2") + ev("3") +
         ev("4") + ev("5") + ev("6") + ev("7") + ev("8") +
         ev("9") + ev("0") + ev("+") + ev("*") + ev("-") +
         ev("%") + ev("=") + ev(">") + ev("<") + ev(">=") +
         ev("<=") + ev("(") + ev(")") + ev("!") + ev("#") +
         ev(";") + ev(".") + ev(" ") + ev(","))) .

        eq applyFunction("DeleteAnyChar",S) = ""   .
endfm
```

```
fmod COPY-FUNCTION is
        inc FUNCTION-AUX .
        var S : String .

        *** To copy anyChar =/= "\n"
        op copy-non-nl : -> Function .
        eq copy-non-nl = ("CopyNonNewLine",
        (ev("a") + ev("b") + ev("c") + ev("d") + ev("e") +
         ev("f") + ev("g") + ev("h") + ev("i") + ev("j") +
         ev("k") + ev("l") + ev("m") + ev("n") + ev("o") +
         ev("p") + ev("q") + ev("r") + ev("s") + ev("t") +
         ev("u") + ev("v") + ev("w") + ev("x") + ev("y") +
         ev("z") + ev("A") + ev("B") + ev("C") + ev("D") +
         ev("E") + ev("F") + ev("G") + ev("H") + ev("I") +
         ev("J") + ev("K") + ev("L") + ev("M") + ev("N") +
         ev("O") + ev("P") + ev("Q") + ev("R") + ev("S") +
         ev("T") + ev("U") + ev("V") + ev("W") + ev("X") +
         ev("Y") + ev("Z") + ev("1") + ev("2") + ev("3") +
         ev("4") + ev("5") + ev("6") + ev("7") + ev("8") +
         ev("9") + ev("0") + ev("+") + ev("*") + ev("-") +
         ev("%") + ev("=") + ev(">") + ev("<") + ev(">=") +
         ev("<=") + ev("(") + ev(")") + ev("!") + ev("#") +
         ev(";") +  ev(".") + ev(" ") + ev("/") + ev(",") )) .
        eq applyFunction("CopyNonNewLine",S) = S .

        *** To copy anyChar =/= "/"
        op copy-non-slashe : -> Function .
        eq copy-non-slashe = ("CopyNonslashe",
        (ev("a") + ev("b") + ev("c") + ev("d") + ev("e") +
         ev("f") + ev("g") + ev("h") + ev("i") + ev("j") +
         ev("k") + ev("l") + ev("m") + ev("n") + ev("o") +
         ev("p") + ev("q") + ev("r") + ev("s") + ev("t") +
         ev("u") + ev("v") + ev("w") + ev("x") + ev("y") +
         ev("z") + ev("A") + ev("B") + ev("C") + ev("D") +
         ev("E") + ev("F") + ev("G") + ev("H") + ev("I") +
         ev("J") + ev("K") + ev("L") + ev("M") + ev("N") +
         ev("O") + ev("P") + ev("Q") + ev("R") + ev("S") +
         ev("T") + ev("U") + ev("V") + ev("W") + ev("X") +
         ev("Y") + ev("Z") + ev("1") + ev("2") + ev("3") +
         ev("4") + ev("5") + ev("6") + ev("7") + ev("8") +
         ev("9") + ev("0") + ev("+") + ev("*") + ev("-") +
         ev("%") + ev("=") + ev(">") + ev("<") + ev(">=") +
         ev("<=") + ev("(") + ev(")") + ev("!") + ev("#") +
         ev(";") +  ev(".") + ev(" ") + ev("\n") + ev(",") )) .
        eq applyFunction("CopyNonslashe",S) = S .
```

```
        *** To copy "\n" from a string
        op copy-nl : -> Function .
        eq copy-nl = ("CopyNewLine",ev("\n")) .
        eq applyFunction("CopyNewLine",S) = S .
endfm

fmod DEL-COMBINATORS is
        inc DREX-SEMANTICS .
        inc DEL-FUNCTION .

        var S : String . vars I J : Nat .

        op DEL-SLASHES : -> Function .
        eq DEL-SLASHES = ("DELETESLASHES",
                domain(split(del-slashes,del-slashes))) .
        eq applyFunction("DELETESLASHES",S) =
         apply(split(del-slashes,del-slashes),S,0,length(S)) .

        op DEL-NON-NL : -> Function .
        eq DEL-NON-NL = ("DELETENONNEWLINE",
                domain(iterate(del))) .
        eq applyFunction("DELETENONNEWLINE",S) =
           apply(iterate(del),S,0,length(S)) .

        op DEL-COMM : -> Function .
        eq DEL-COMM = ("DELETECOMM",
           domain(split(DEL-SLASHES,DEL-NON-NL))) .
        eq applyFunction("DELETECOMM",S) =
         apply(split(DEL-SLASHES,DEL-NON-NL),S,0,length(S)) .

        op DEL-COMM-LINE : -> Function .
        eq DEL-COMM-LINE = ("DELCOMMLINE",
                        domain(split(DEL-COMM,del-nl))) .
        eq applyFunction("DELCOMMLINE",S) =
         apply(split(DEL-COMM,del-nl),S,0,length(S)) .
endfm

fmod COPY-COMBINATORS is
        inc DREX-SEMANTICS .
        inc COPY-FUNCTION .

        var S : String . vars I J : Nat .

        op COPY-NON-NL : -> Function .
        eq COPY-NON-NL = ("COPYNONNL",
          domain(iterate(copy-non-nl))) .
```

```
        eq applyFunction("COPYNONNL",S) =
         apply(iterate(copy-non-nl),S,0,length(S)) .

        op COPY-TXT : -> Function .
        eq COPY-TXT = ("COPYTXT",
          domain(split(copy-non-slashe,COPY-NON-NL))) .
        eq applyFunction("COPYTXT",S) =
         apply(split(copy-non-slashe,COPY-NON-NL),
               S,0,length(S)) .

        op COPY-LINE : -> Function .
        eq COPY-LINE = ("COPYLINE",
         domain(conditional(copy-nl,
                  split(COPY-TXT,copy-nl)))) .
        eq applyFunction("COPYLINE",S) =
         apply(conditional(copy-nl,
               split(COPY-TXT,copy-nl)),S,0,length(S)) .
endfm

fmod CODE-TRANSFORMATION is
        inc DEL-COMBINATORS .
        inc COPY-COMBINATORS .
        inc INT .

        vars S S' : String . var M    : Nat .

        op process-line : -> Function .
        eq process-line = ("ProcessLine",
           domain(conditional(DEL-COMM-LINE,COPY-LINE))) .
        eq applyFunction("ProcessLine",S) =
         apply(conditional(DEL-COMM-LINE,COPY-LINE),
               S,0,length(S)) .

        op last-line : -> Function .
        eq last-line = ("LastLine",
          domain(conditional(DEL-COMM,COPY-TXT))) .
        eq applyFunction("LastLine",S) =
         apply(conditional(DEL-COMM,COPY-TXT),
                S,0,length(S)) .

        op process-last-line : -> Function .
        eq process-last-line = ("ProcessLastLine",
          domain(conditional(process-line,last-line))) .
        eq applyFunction("ProcessLastLine",S) =
         apply(conditional(process-line,last-line),
               S,0,length(S)) .
```

```
        op process-lines : -> Function .
        eq process-lines = ("ProcessLines",
           domain(iterate(process-line))) .
        eq applyFunction("ProcessLines",S) =
         apply(iterate(process-line),S,0,length(S)) .

        op delete-comm : String -> String .
        ceq delete-comm(S) =
             applyFunction("ProcessLastLine",S)
                  if find(S,"\n",0) == notFound .
        ceq delete-comm(S) =
             applyFunction("ProcessLastLine",S)
                  if (length(S) - find(S,"\n",0))== 1 .
        ceq delete-comm(S) =
             applyFunction("ProcessLine",S') +
             delete-comm(substr(S,M + 1,length(S)))
                  if M := find (S,"\n",0)
                   /\ S' := substr(S,0,M + 1) [owise] .
endfm
```

APPENDIX C
NDREX Specification in Maude

```
*** Semantics of the split combinator

mod SPLIT-SEMANTICS is
inc SEM-INTERFACE .
pr POS-LIST .
vars S S1 S2 S'   : String .
vars E1 E2        : Expr .
vars BinD1 BinD2  : Bool .
vars K I J        : Nat .
vars L1 L2        : List .

*** Domain of a split expression
eq domain(split(E1,E2)) = domain(E1) domain(E2) .

*** Semantics of split
crl [splitRule] : apply(split(E1,E2),S,I,J)
=> S1 + S2
if chkIndex(S,I,J)
/\ S' := substr(S,I, sd(J, I))
/\ L1, K, L2 := splits(S',E1,E2)
/\ S1 := apply(E1, S', 0, K + 1)
/\ S2 := apply(E2, S', K + 1, length(S')) .

*** The split position(s) in
*** a string given two expressions
op splits : String Expr Expr -> Nat .
eq splits(S, E1, E2) = splitsPosition(S, 0, E1, E2) .

*** a helper function for the
*** split position(s) algorithm
op splitsPosition : String Nat Expr Expr -> List .
ceq splitsPosition(S,K,E1,E2) = mt if K > length(S) .
ceq splitsPosition(S,K,E1,E2)
= if BinD1 and BinD2 then
K, splitsPosition(S, K + 1,E1, E2)
else
splitsPosition(S, K + 1, E1, E2)        fi
if K <= length(S)
/\ BinD1 := chkDomain(E1,substr(S,0,K + 1))
```

```
/\ BinD2 := chkDomain(E2, substr(S,K + 1,length(S))) .
endm

*** DReX semantics

mod DREX-SEMANTICS is
pr BASE-SEMANTICS .
pr COMBINE-SEMANTICS .
pr COND-SEMANTICS .
pr ITER-SEMANTICS .
pr SPLIT-SEMANTICS .
endm
```

Curriculum Vitae

Personal Information

- Name : Shadi Ayman Alhaj.
- Date and Place of birth : 1989 Amman – Jordan.
- Marital Status : Single.
- Nationality : Jordanian.
- Address : Dhahran, Kingdom of Saudi Arabia.
- Telephone : 00966-532449502.
- E-mail : shadi89_alhaj@yahoo.com.

Career Objective:

To build a career effort, experience and to find a position that would allow growth and provide opportunities of experience.

Qualifications:

2015-2017:

University : King Fahd University of Petroleum and Minerals.
Faculty : Information and Computer Science.
Degree : Master
Major : Computer Science.
GPA : 3.53 out 4.0.

2010-2013:

University : Hashemite University.
Faculty : Prince Al-Hussein Bin Abdallah II for Information Technology
Degree : Bachelor
Major : Computer Science.
GPA : 3.31 out 4.0.

2007-2009:

College : Amman Training Center.
Faculty : Information Technology.
Degree : Diploma.
Major : Information Technology.
GPA : 81 out 100.

Work Experience:

Position	: Oracle Developer.
Employer	: Future Applied Computer Technology.
Period	: Apr. 2014 to Jan.2015.
Position	: Operator at Data Center.
Employer	: Jordan Kuwait Bank.
Period	: Sep. 2009 to Apr. 2014.
Position	: Trainee.
Employer	: Palco.
Period	: Jul. 2013 to Aug. 2013.
Position	: Trainee.
Employer	: Oracle.
Period	: Jun. 2013 to Jul. 2013.

Programming Language

- Oracle Developer Forms and Reports.
- SQL.
- PL/SQL.
- Java.
- Visual Basic 2013.
- C++.

Short Courses

- Introduction to SAP HANA Database.
- Introduction to SAP Navigation and Basics.

Skills:

- Organized and excellent in communications with team work.
- Hard working and ability to work under pressure.
- Flexibility and creativity.

Languages

- Have a good command of spoken and written in Arabic and English.

Publication

- Multi-layers Video Steganography: A Novel Technique for Image Hiding SA Alhaj, AM Shaheen, TM Alkharobi, Transactions on Networks and Communications 4 (6), 53.